Design and implementation of real time computer vision algorithms for video surveillance applications

Hicham Ghorayeb

Collège doctoral

N° attribué par la bibliothèque

__/__/__/__/__/__/__/__/__/__/

T H E S E

pour obtenir le grade de
Docteur de l'Ecole des Mines de Paris
Spécialité «Informatique temps réel – Robotique - Automatique»

présentée et soutenue publiquement
par
Hicham GHORAYEB

le 12/ 09/2007

CONCEPTION ET MISE EN ŒUVRE D'ALGORITHMES DE VISION TEMPS REEL POUR LA VIDEO SURVEILLANCE INTELLIGENTE

Directeur de thèse : Claude LAURGEAU

Jury

M. Patrick MEYRUEIS (Univ. Strasbourg)	Rapporteur
M. Patrick SIARRY (Paris XII)	Rapporteur
M. Mohamed AKIL (ESIEE)	Examinateur
M. Bruno STEUX (EMP)	Examinateur
M. Claude LAURGEAU (EMP)	Examinateur
M. Fernand MEYER (EMP)	Examinateur
M. Karl SCHWERDT (VIDATIS)	Examinateur

Á
Hussein et Najah
mes parents

Remerciements

Cette thèse n'aurait vu le jour sans la confiance, la patience et la générosité de mon directeur de thèse, Monsieur Claude Laurgeau, directeur du centre de robotique de l'Ecole des Mines de Paris. Je voudrais aussi le remercier pour le temps et la patience qu'il m'a accordé tout au long de ces années, d'avoir cru en mes capacités et de m'avoir fourni d'excellentes conditions logistiques et financières.

Mes plus sincères remerciements vont également à Monsieur Bruno Steux, qui en agissant à titre d'encadrant a fortement enrichi ma formation. Ses conseils techniques et ses commentaires auront été fort utiles. De plus, les conseils qu'il m'a divulgu'e tout au long de la rédaction, ont toujours été clairs et succincts, me facilitant grandement la tâche et me permettant d'aboutir à la production de cette thèse.

J'aimerais par ailleurs souligner la collaboration importante avec Monsieur Mohamed Akil, professeur à l'ESIEE. Notre collaboration dans le cadre de l'accélération des détecteurs visuels d'objets sur des cartes graphiques et des cartes FPGA. Dans le cadre de cette collaboration j'ai encadré trois groupes d'étudiants de l'ESIEE dans leurs stages. Les stages ont aboutit à des prototypes de validation de l'idée.

Je voudrais également remercier de tout mon coeur Monsieur Fawzi Nashashibi, pour ses précieux conseils et son support moral tout le long de la thèse.

Je remercie Monsieur Patrick Siarry, professeur á l'université Paris XII, pour l'intérrêt qu'il a apporté à mon travail en me consacrant de son temps pour rapporter sur le présent travail.
Je remercie également Monsieur Patrick Meyrueis, professeur à l'université de Strasbourg, pour la rapidité avec laquelle il m'a accordé du temps pour rapporter sur le présent travail.

Je remercie Monsieur Karl Schwerdt, et monsieur Didier Maman, fondateurs de la société VIDATIS pour leur collaboration dans le co-encadrement du stage de Khattar Assaf.

Je remercie les stagiaires avec qui j'ai travaillé: Khattar Assaf, Hamid Medjahed, et Marc-andré Lureau.

Enfin une pensée émue pour mes amis avec qui j'ai partagé la L007 pendant ces années de thèse: Jacky, Clement, Jorge, Marc et Amaury. Et n'oubliant pas les autres collègues dans l'aquarium: Safwan, Xavier, Rasul et Laure. Et surtout, ceux qui sont dans des bureaux avec une salle de réunion: Fawzi, Francois, Samer, Iyad et Ayoub. Et les collègues qui sont dans des bureaux éparpillès comme Bogdan.

Résumé

Mots clés: *Reconnaissance des formes, analyse vidéo intélligente, systèmes de transport intélligents, vidéo surveillance intelligente, GPGPU, apprentissage automatique, boosting, LibAdaBoost, soustraction du fond adaptative, projet PUVAME.*

Ce travail de thèse a pour sujet l'étude des algorithmes de vision temps réel pour l'analyse vidéo intelligente. Ces travaux ont eut lieu au laboratoire de robotique de l'Ecole des Mines de Paris (CAOR).
Notre objectif est d'étudier les algorithmes de vision utilisés aux différents niveaux dans une chaîne de traitement vidéo intelligente. On a prototypé une chaîne de traitement générique dédiée à l'analyse du contenu du flux vidéo. En se basant sur cette chaîne de traitement, on a développé une application de détection et de suivi de piétons. Cette application est une partie intégrante du projet PUVAME. Ce projet a pour objectif d'améliorer la sécurité des personnes dans des zones urbaines près des arrêts de bus.
Cette chaîne de traitement générique est composée de plusieurs étapes : d'une part détection d'objets, classification d'objets et suivi d'objets. D'autres étapes de plus haut niveau sont envisagées comme la reconnaissance d'actions, l'identification, la description sémantique ainsi que la fusion des données de plusieurs caméras. On s'est intéressé aux deux premières étapes. On a exploré des algorithmes de segmentation du fond dans un flux vidéo avec caméra fixe. On a implémenté et comparé des algorithmes basés sur la modélisation adaptative du fond.

On a aussi exploré la détection visuelle d'objets basé sur l'apprentissage automatique en utilisant la technique du boosting. Cependant, On a développé une librairie intitulée LibAdaBoost qui servira comme un environnement de prototypage d'algorithmes d'apprentissage automatique. On a prototypé la technique du boosting au sein de cette librairie. On a distribué LibAdaBoost sous la licence LGPL. Cette librarie est unique avec les fonctionnalités qu'elle offre.

On a exploré l'utilisation des cartes graphiques pour l'accélération des algorithmes de vision. On a effectué le portage du détecteur visuel d'objets basé sur un classifieur généré par le boosting pour qu'il s'exécute sur le processeur graphique. On était les premiers à effectuer ce portage. On a trouvé que l'architecture du processeur graphique est la mieux adaptée pour ce genre d'algorithmes.

La chaîne de traitement a été implémentée et intégrée à l'environnement RTMaps. On a évalué ces algorithmes sur des scénarios bien définis. Ces scénarios ont été définis dans le cadre de PUVAME.

Abstract

Keywords: *Intelligent video analysis, intelligent transportation systems, intelligent video surveillance, GPGPU, boosting, LibAdaBoost, adaptive background subtraction, PUVAME project.*

In this dissertation, we present our research work held at the Center of Robotics (CAOR) of the *Ecole des Mines de Paris* which tackles the problem of intelligent video analysis.

The primary objective of our research is to prototype a generic framework for intelligent video analysis. We optimized this framework and configured it to cope with specific application requirements. We consider a people tracker application extracted from the PUVAME project. This application aims to improve people security in urban zones near to bus stations.
Then, we have improved the generic framework for video analysis mainly for background subtraction and visual object detection. We have developed a library for machine learning specialized in boosting for visual object detection called LibAdaBoost. To the best of our knowledge LibAdaBoost is the first library in its kind. We make LibAdaBoost available for the machine learning community under the LGPL license.

Finally we wanted to adapt the visual object detection algorithm based on boosting so that it could run on the graphics hardware. To the best of our knowledge we were the first to implement visual object detection with sliding technique on the graphics hardware. The results were promising and the prototype performed three to nine times better than the CPU.

The framework was successfully implemented and integrated to the RTMaps environment. It was evaluated at the final session of the project PUVAME and demonstrated its fiability over various test scenarios elaborated specifically for the PUVAME project.

Contents

Part I

Introduction and state of the art

Chapter 1

French Introduction

Contents

Pour accentuer la bonne lisibilité de cette thèse, et pour permettre au lecteur de se créer une vision globale du contenu avant même d'entamer la lecture, nous voudrions brièvement exposer à cette place les points clés de la thèse acompagnés de quelques remarques. Ces quelques lignes vont, comme nous le croyons, servir à une meilleure orientation dans l'ensemble de ce traité.

Cette thèse étudie les algorithmes de vision temps réel pour l'analyse vidéo intelligente. Ces travaux ont eut lieu au laboratoire de robotique de l'Ecole des Mines de Paris (CAOR).
Notre objectif est d'étudier les algorithmes de vision utilisés aux différents niveaux dans une chaîne de traitement vidéo intelligente. On a prototypé une chaîne de traitement générique dédiée à l'analyse du contenu du flux vidéo. En se basant sur cette chaîne de traitement, on a développé une application de détection et de suivi de piétons. Cette application est une partie intégrante du projet PUVAME. Ce projet a pour objectif d'améliorer la sécurité des personnes dans des zones urbaines près des arrêts de bus.
Cette chaîne de traitement générique est composée de plusieurs étapes : d'une part détection d'objets, classification d'objets et suivi d'objets. D'autres étapes de plus haut niveau sont envisagées comme la reconnaissance d'actions, l'identification, la description sémantique ainsi que la fusion des données de plusieurs caméras. On s'est intéressé aux deux premières étapes. On a exploré des algorithmes de segmentation du fond dans un flux vidéo avec caméra fixe. On a implémenté et comparé des algorithmes basés sur la modélisation adaptative du fond.

On a aussi exploré la détection visuelle d'objets basé sur l'apprentissage automatique

en utilisant la technique du boosting. Cependant, On a développé une librairie intitulée LibAdaBoost qui servira comme un environnement de prototypage d'algorithmes d'apprentissage automatique. On a prototypé la technique du boosting au sein de cette librairie. On a distribué LibAdaBoost sous la licence LGPL. Cette librarie est unique avec les fonctionnalités qu'elle offre.

On a exploré l'utilisation des cartes graphiques pour l'accéléraction des algorithmes de vision. On a effectué le portage du détecteur visuel d'objets basé sur un classifieur généré par le boosting pour qu'il s'exécute sur le processeur graphique. On était les premiers à effectuer ce portage. On a trouvé que l'architecture du processeur graphique est la mieux adaptée pour ce genre d'algorithmes.

La chaîne de traitement a été implementée et integrée à l'environnement RTMaps. On a évalué ces algorithmes sur des scénarios bien définis. Ces scénarios ont été définis dans le cadre de PUVAME.

Le document est divisé en quatre parties principales derrière lesquelles nous rajoutons une conclusion générale suivie par les annexes:

1. Nous commençons dans Chapitre 2 par une introduction en anglais du manuscrit. Chapitre 3 prèsente une discussion générale sur les systèmes d'analyse vidéo intelligente pour la vidéo surveillance.

2. Dans Chapitre 4 on décrit une chaîne de traitement générique pour l'analyse vidéo intelligent. Cette chaîne de traitement pourra être utilisée pour prototyper des applications de vidéo surveillance intelligente. On commence par la description de certaines parties de cette chaîne de traitement dans lec chapitres qui suivent: on discute dans Chapitre 5 des différentes implémentations pour l'algorithme de modélisation adaptative du fond pour la ségmentation des objets en mouvement dans une scène dynamique avec caméra fixe. Dans Chapitre 6 on décrit une technique d'apprentissage automatique nommée boosting qui est utilisée pour la détection visuelle d'objets. Le suivi d'objets est introduit dans le Chapitre 7.

3. Dans Chapitre 8 on présente la programmation par flux de données. On présente ainsi l'architecture des processeurs graphiques. La modélisation en utilisant le modèle par flux de données est décrite dans Chapitre 9.

4. On décrit le projet PUVAME[1] dans Chapitre 10.

5. Finalement, on décrit dans Chapitre 11 la conclusion de nos traveaux, et notre vision sur le futur travail à munir. Une conclusion en français est fournie dans Chapitre 12.

[1]Protection des Vulnérables par Alarmes ou Manoeuvres

1.1 Algorithmes

Cette patie est consacrée, en cinq chapitres, aux algorithmes destinés à être constitutifs d'une plateforme générique pour la vidéo surveillance.

L'organisation opérationnelle d'une plateforme logicielle pour la vidéo surveillance est d'abord présentée dans un chapitre. La détection d'objets en mouvement est ensuite traitée dans le chapitre suivant avec des méthodes statistiques et d'autres non linéaires. Des algorithmes sont présentés basés sur une approche adaptative d'extraction de l'arrière plan. La détection d'objets est dépendante de la scène et doit prendre en compte l'ombrage et le changement de luminosité. La question des fausses alarmes se pose pour renforcer la robustesse. A cet effet, deux approches sont traitées, l'approche statistique non paramétrable et l'approche déterministe. L'évaluation des algorithmes considérés se base sur des métriques prises en compte dans deux cas de scènes: un interne et un externe.

L'apprentissage pour la détection d'objets est considérée dans un avant dernier chapitre. Des algorithmes de Boosting sont introduits notamment AdaBoost avec un Algorithme génétique pour l'apprentissage faible. Un ensemble de classifieurs faibles est étudié et implémenté allant des classifieurs rectangulaires de Viola and Jones aux classifieurs de points de control et d'autres.

On présente la librairie logicielle LibAdaBoost. LibAdaBoost constitue un environnement unifié pour le boosting. Cet environnement est ouvert et exploitable par ceux ayant à valider des algorithmes d'apprentissage.

Le dernier chapitre traite du tracking d'objets avec une méthode Monte Carlo Tracking. La méthode est robuste sous différentes situations d'illumination et est renforcée par l'intégration du détecteur visuel par boosting.

1.2 Architecture

Cette partie en deux chapitres propose une application temps réel de la détection de visage avec implémentation spécifique pour accéleration des algorithmes sur processeur graphique. L'accent est mis sur l'implémentation GPU, travail de recherche en plein développement et notemment pour paralléliser et accélérer des algorithmes généraux sur un proceseur graphique. Après une description succinte et pédagogique aussi bien de l'architecture du GPU, de son modèle de programmation que des logiciels de programmation des GPUs, on décrit avec détails: l'omplémentation, les optimisations, les différentes stratégies possibles de partitionnement de l'application entre CPU et GPU. La solution adoptée est d'implanter le classifieur sur le GPU, les autres étapes de l'application sont programmées sur le CPU. Les résultats obtenus sont très prometteurs et permettent d'accélérer d'un facteur 10.

1.3 Application

Cette partie applicative est consacrée à une application des traveaux de la thèse dans le cadre du projet PUVAME, qui vise à concevoir une alarme de bord pour alerter les chauffeurs de bus, en cas de risque de collision avec des piétons, aux abords des stations d'autobus et aux intersections.

Chapter 2

Introduction

Contents

While researching the materials presented in this thesis I have had the pleasure of covering a wider range of the discipline of Computer Science than most graduate students.

My research began with the european project CAMELLIA[1]. CAMELLIA aims to prototype a smart imaging core dedicated for mobile and automotive domains. I was in charge of the implementation of image stabilization and motion estimation algorithms [SAG03a]. As well as the design of a motion estimator core [SAG03b]. I was soon drawn into studying the details of data parallel computing on stream architectures and more specially on graphics hardware. I focused also on intelligent video analysis with machine learning tools.

In this dissertation, we are primarily concerned with studying intelligent video analysis framework. We focused on object detection and background segmentation. We explored the usage of graphics hardware for accelerating computer vision algorithms as low cost solution. We evaluated our algorithms in several contexts: video surveillance and urban road safety.

2.1 Contributions

This dissertation makes several contributions to the areas of computer vision and general purpose computation on graphics hardware:

- We have evaluated several implementations of state of the art adaptive background subtraction algorithms. We have compared their performance and we have tried to improve these algorithms using high level feedback.

[1] Core for Ambient and Mobile intELLigent Imaging Applications

- We present boosting for computer vision community as a tool for building visual detectors. The pedagogic presentation is based on boosting as a machine learning framework. Thus, we start from abstract machine learning concepts and we develop computer vision concepts for features, finders and others.

- We have also developed a software library called LibAdaBoost. This library is a machine learning library which is extended with a boosting framework. The boosting framework is extended to support visual objects. To the best of our knowledge this is the first library specialized in boosting and which provides both: a SDK for the development of new algorithms and a user toolkit for launching learnings and validations.

- We present a streaming formulation for visual object detector based on visual features. Streaming is a natural way to express low-level and medium-level computer vision algorithms. Modern high performance computing hardware is well suited for the stream programming model. The stream programming model helps to organize the visual object detector computation optimally to execute on graphics hardware.

- We present an integration of intelligent video analysis block within a large framework for urban area safety.

Efficient implementation of low-level computer vision algorithms is an active research area in the computer vision community, and is beyond the scope of this thesis. Our analysis started from a general framework for intelligent video analysis. We factorized a set of algorithms at several levels: pixel-level, object-level and frame-level. We focused on pixel-level algorithms and we integrated these algorithms to a generic framework. In the generic framework we developed state of the art approaches to instantiate the different stages.

2.2 Outline

We begin in Chapter 3 with a background discussion of Intelligent Video Surveillance Systems.

In Chapter 4 we describe a generic framework for intelligent video analysis that can be used in a video surveillance context. We started to detail some stages of this generic framework in the next chapters: we discuss in Chapter 5 different implementations of adaptive background modeling algorithms for moving object detection. In Chapter 6 we describe machine learning for visual object detection with boosting. Object tracking is discussed in Chapter 7.

In Chapter 8 we present stream programming model, and programmable graphics hardware. Streaming formulation and implementation for visual object detection is described in Chapter 9.

We describe the PUVAME[2] application in Chapter 10.

Finally, we suggest areas for future research, reiterate our contributions, and conclude this dissertation in Chapter 11.

This thesis is a testament to the multi-faceted nature of computer science and the many and varied links within this relatively new discipline. I hope you will enjoy reading it as much as I enjoyed the research which it describes.

[2]Protection des Vulnérables par Alarmes ou Manoeuvres

Chapter 3

Intelligent video surveillance systems (IVSS)

Contents

3.1 What is IVSS?

3.1.1 Introduction

Intelligent video surveillance addresses the use of automatic video analysis technologies in video surveillance applications. This chapter attempts to answer a number of questions about intelligent surveillance: what are the applications of intelligent surveillance? what are the system architectures for intelligent surveillance? what are the key technologies? and what are the key technical challenges?

3.1.2 Historical background

It is because of the advance in computing power, availability of large-capacity storage devices and high-speed network infrastructure, that we find inexpensive multi sensor video surveillance systems. Traditionally, the video outputs are processed online by human operators and are usually saved to tapes for later use only after a forensic event. The increase in the number of cameras in ordinary surveillance systems overloaded both the number of operators and the storage devices and made it impossible to ensure proper monitoring of sensitive areas for long times. In order to filter out redundant information and increase the response time to forensic events, helping the human operators by detecting important events in video by the use of smart video surveillance systems has become an important requirement.

In the following sections we group video surveillance systems into three generations as in [Ded04].

- (1GSS, 1960-1980) were based on analog sub systems for image acquisition, transmission and processing. They extended human eye in spatial sense by transmitting the outputs of several cameras monitoring a set of sites to the displays in a central control room. They had major drawbacks like requiring high bandwidth, difficult archiving and retrieval of events due to large number of video tape requirements and difficult online event detection which only depended on human operators with limited attention span.

- (2GSS, 1980-2000) The next generation surveillance systems were hybrids in the sense that they used both analog and digital sub systems to resolve some drawbacks of their predecessors. They made use of the early advances in digital video processing methods that provide assistance to the human operators by filtering out spurious events. Most of the work during 2GSS is focused on real time event detection.

- (3GSS, 2000-) Third generation surveillance systems provide end-to-end digital systems. Image acquisition and processing at the sensor level, communication through mobile and fixed heterogeneous broadband networks and image storage at the central servers benefit from low cost digital infrastructure. Unlike previous generations, in 3GSS some part of the image processing is distributed

toward the sensor level by the use of intelligent cameras that are able to digitize and compress acquired analog image signals and perform image analysis algorithms.

Third generation surveillance systems (3GSS) assist human operators with online alarm generation defined on complex events and with offline inspection as well. 3GSS also handle distributed storage and content-based retrieval of video data. It can be used both for providing the human operator with high level data to help him to make the decisions more accurately and in a shorter time and for offline indexing and searching stored video data effectively. The advances in the development of these algorithms would lead to breakthroughs in applications that use visual surveillance.

3.2 Applications of intelligent surveillance

In this section we describe a few applications of smart surveillance technology [HBC+03]. We group the applications into three broad categories: real time alarms, automatic forensic video retrieval, and situation awareness.

3.2.1 Real time alarms

We can define two types of alerts that can be generated by a smart surveillance system, user defined alarms and automatic unusual activity alarms.

3.2.2 User defined alarms

In this type the system detects a variety of user defined events that occur in the monitored space and notifies the user in real time. So it is up to the user to evaluate the situation and to take preventive actions. Some typical events are presented.

Generic alarms

These alerts depend on the movement properties of objects in the monitored space. Following are a few common examples:

- Motion Detection: This alarms detects movement of any object within a specified zone.
- Motion Characteristic Detection: These alarms detect a variety of motion properties of objects, including specific direction of object movement (entry through exit lane), object velocity bounds checking (object moving too fast).
- Abandoned Object alarm: This detects objects which are abandoned, e.g., a piece of unattended baggage in an airport, or a car parked in a loading zone.
- Object Removal: This detects movements of a user-specified object that is not expected to move, for example, a painting in a museum.

Class specific alarms

These alarms use the type of object in addition to its movement properties. As examples:

- Type Specific Movement Detection: Consider a camera that is monitoring runways at an airport. In such a scene, the system could provide an alert on the presence or movement of people on the tarmac but not those of aircrafts.
- Statistics: Example applications include, alarms based on people counts (e.g., more than one person in security locker) or people densities (e.g., discotheque crowded beyond an acceptable level).

Behavioral alarms

when detecting adherence to, or deviation from, learnt models of motion patterns. Such models are based on training and analyzing movement patterns over extended periods of time. These alarms are used in specific applications and use a significant amount of context information, for example:

- Detecting shopping groups at retail checkout counters, and alerting the store manager when the length of the queue at a counter exceeds a specified number.
- Detecting suspicious behavior in parking lots, for example, a person stopping and trying to open multiple cars.

High Value Video Capture

This is an application which augments real time alarms by capturing selected clips of video based on pre-specified criteria. This becomes highly relevant in the context of smart camera networks, which use wireless communication.

3.2.3 Automatic unusual activity alarms

Unlike the user-defined alarms, here the system generates alerts when it detects activity that deviates from the norm. The smart surveillance system achieves this based on learning normal activity patterns. For example: when monitoring a street the system learns that vehicles move about on the road and people move about on the side walk. Based on this pattern the system will provide an alarm when a car drives on the sidewalk. Such unusual activity detection is the key to effective smart surveillance, as the user cannot manually specify all the events of interest.

3.2.4 Automatic Forensic Video Retrieval (AFVR)

The capability to support forensic video retrieval is based on the rich video index generated by automatic tracking technology. This is a critical value-add from using

smart surveillance technologies. Typically the index consists of such measurements as object shape, size and appearance information, temporal trajectories of objects over time, object type information, in some cases specific object identification information. In advanced systems, the index may contain object activity information.

During the incident the investigative agencies had access to hundreds of hours of video surveillance footage drawn from a wide variety of surveillance cameras covering the areas in the vicinity of the various incidents. However, the task of manually sifting through hundreds of hours of video for investigative purposes is almost impossible. However if the collection of videos were indexed using visual analysis, it would enable the following ways of retrieving the video

- Spatio-Temporal Video Retrieval: An example query in this class would be, *Retrieve all clips of video where a blue car drove in front of the 7/11 Store on 23rd street between the 26th of July 2pm and 27th of July 9am at speeds > 25mph.*

3.2.5 Situation awareness

To ensure a total security at a facility requires systems that continuously track the identity, location and activity of objects within the monitored space. Typically surveillance systems have focused on tracking location and activity, while biometrics systems have focused on identifying individuals. As smart surveillance technologies mature, it becomes possible to address all these three key challenges in a single unified framework giving rise to, joint location identity and activity awareness, which when combined with the application context becomes the basis for situation awareness.

3.3 Scenarios and examples

Below are some scenarios that smart surveillance systems and algorithms might handle:

3.3.1 Public and commercial security

- Monitoring of banks, department stores, airports, museums, stations, private properties and parking lots for crime prevention and detection
- Patrolling of highways and railways for accident detection
- Surveillance of properties and forests for fire detection
- Observation of the activities of elderly and infirm people for early alarms and measuring effectiveness of medical treatments
- Access control

3.3.2 Smart video data mining

- Measuring traffic flow, pedestrian congestion and athletic performance
- Compiling consumer demographics in shopping centers and amusement parks
- Extracting statistics from sport activities
- Counting endangered species
- Logging routine maintenance tasks at nuclear and industrial facilities
- Artistic performance evaluation and self learning

3.3.3 Law enforcement

- Measuring speed of vehicles
- Detecting red light crossings and unnecessary lane occupation

3.3.4 Military security

- Patrolling national borders
- Measuring flow of refugees
- Monitoring peace treaties
- Providing secure regions around bases
- Assisting battlefield command and control

3.4 Challenges

Probably the most important types of challenges in the future development of smart video surveillance systems are:

3.4.1 Technical aspect

A large number of technical challenges exists especially in visual analysis technologies. These include challenges in robust object detection, tracking objects in crowded environments, challenges in tracking articulated bodies for activity understanding, combining biometric technologies like face recognition with surveillance to generate alerts.

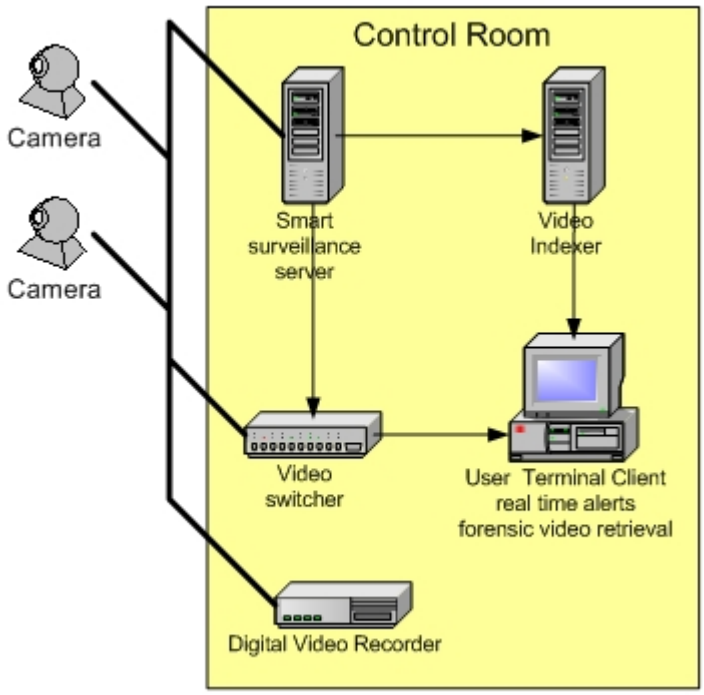


Figure 3.1: Block diagram of a basic surveillance system architecture

3.4.2 Performance evaluation

This is a very significant challenge in smart surveillance system. To evaluate the performance of video analysis systems a lot of annotated data are required. Plus it is a very expensive and tedious process. Also, there can be significant errors in annotation. Therefore it is obvious that these issues make performance evaluation a significant challenge

3.5 Choice of implementation methods

A surveillance system depends on the subsequent architecture: sensors, computing units and media of communication. In the near future, the basic surveillance system architecture presented in Figure 3.5 is likely to be the most ubiquitous in the near future. In this architecture, cameras are mounted in different parts in a facility, all of which are connected to a central control room. The actual revolution in the industry of cameras and the improvements in the media made a revolution in the architectures and open doors for new applications (see Figure 3.5).

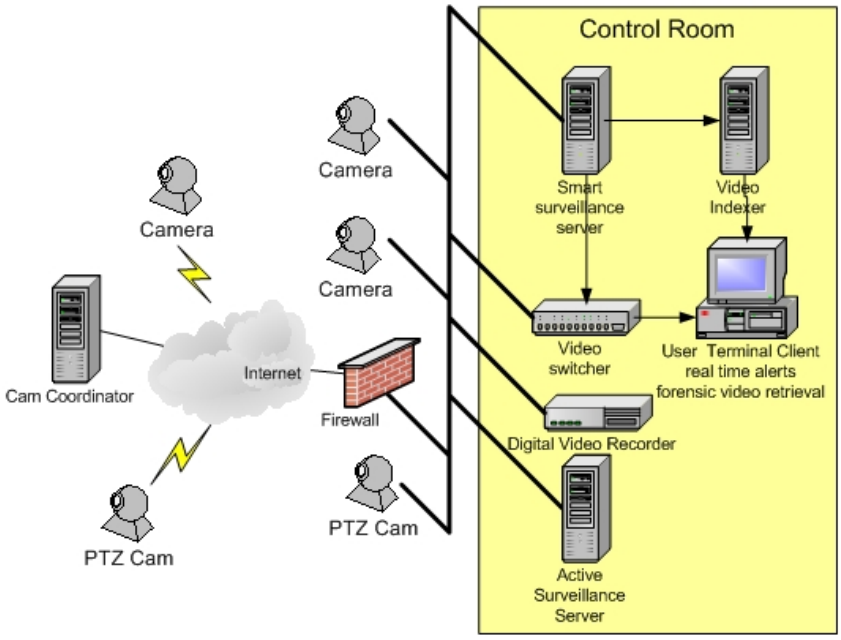


Figure 3.2: Block diagram of an advanced surveillance system architecture

3.6 Conclusion

In this chapter we presented an overview of intelligent video surveillance systems. It consists in an architecture based on cameras and computing servers and visualization terminals. The surveillance application is built upon this architecture and is adapted to the monitored scene domain. In the next chapters we consider a basic intelligent surveillance architecture and the subsequent algorithms are easily transported to more advanced architectures as described in the next paragraph.

Surveillance applications can be described formally regardless the distribution on the computing units (servers or embedded processors in the case of smart cameras). We can abstract this notion using a high level application prototyping system as RTMaps. An application consists in a diagram of components connected to each other. Cameras are represented by source components and servers as processing components, and all visualization and archiving tasks are represented as sink components. Once the application is prototyped it is easy to do the implementation on the real architecture and to distribute the processing modules to the different computing units.

Part II

Algorithms

Chapter 4

Generic framework for intelligent visual surveillance

Contents

The primary objective of this chapter is to give a general overview of the overall process of an intelligent visual surveillance system. Fig. 4 shows the general framework of visual surveillance in dynamic scenes. The prerequisites for effective automatic surveillance using a single camera include the following stages: object detection, object classification, object tracking, object identification, action recognition and semantic description. Although, some stages require interchange of information with other stages, this generic framework provides a good structure for the discussion throughout this chapter. In order to extend the surveillance area and overcome occlusion, fusion of data from multiple cameras is needed. This fusion can involve all the above stages. This chapter is organized as follows: section 4.1 describes the object detection stage. Section 4.2 introduces the object classification stage. Object tracking is presented in section 4.3. Section 4.4 presents action recognition stage. Section 4.6 describes the personal identification stage. Section 4.7 describes the fusion of data from multiple cameras stage.

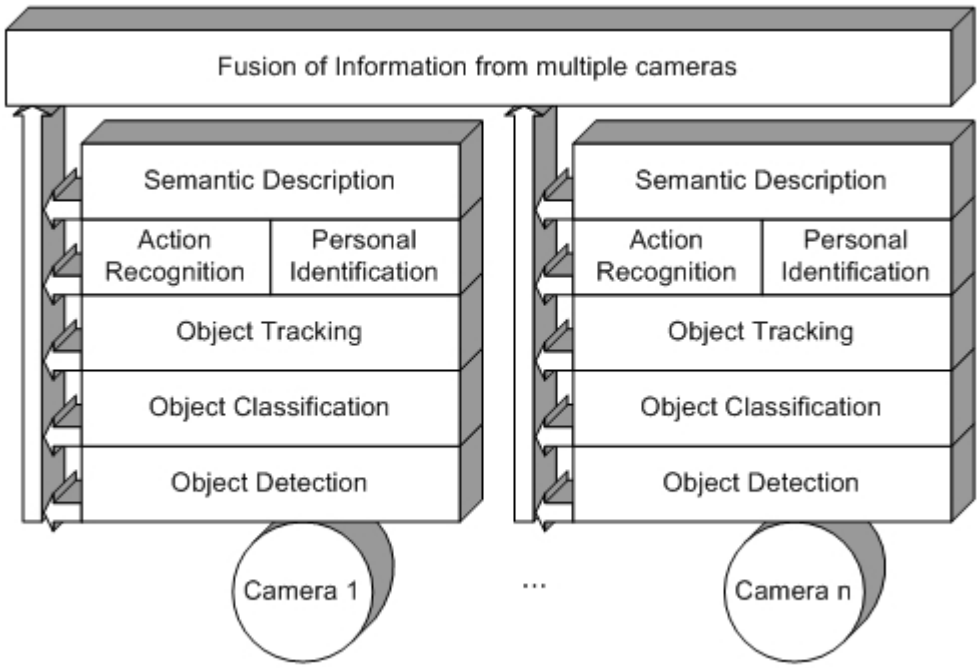


Figure 4.1: A generic framework for intelligent visual surveillance

4.1 Object detection

The first step in nearly every visual surveillance system is moving object detection. Object detection aims at segmenting regions corresponding to moving objects from the rest of the image. Subsequent processes such as classification and tracking are greatly dependent on it. The process of object detection usually involves background modeling and motion segmentation, which intersect each other during processing. Due to dynamic changes in natural scenes such as sudden illumination and weather changes, repetitive motions that cause clutter, object detection is a difficult problem to process reliably.

4.2 Object classification

In real-world scenes, different moving objects may correspond to different moving targets. For instance, the image sequences captured by surveillance cameras mounted in outdoor parking scenes probably include humans, vehicles and other moving objects such as flying birds and moving clouds, etc. It is very important to recognize the type of a detected object in order to track reliably and analyze its activities correctly. Object classification can be considered as a standard pattern recognition issue. At present, there are two major categories of approaches toward moving object classi-

fication which are shape-based and motion-based methods [WHT03]. Shape-based classification makes use of the objects'2D spatial information whereas motion-based classification uses temporally tracked features of objects for the classification solution.

4.3 Object tracking

After object detection, visual surveillance systems generally track moving objects from one frame to another in an image sequence. The objective of tracking is to establish correspondence of objects and object parts between consecutive frames of video stream. The tracking algorithms usually have considerable intersection with lower stages (object detection) and higher stages (action recognition) during processing: they provide cohesive temporal data about moving objects which are used both to enhance lower level processing such as object detection and to enable higher level data extraction such as action recognition.

Tracking in video can be categorized according to the needs of the applications it is used in or according to the methods used for its solution. For instance, in the case of human tracking, whole body tracking is generally adequate for outdoor video surveillance whereas objects' part tracking is necessary for some indoor surveillance and higher level action recognition applications. There are two common approaches in tracking objects as a whole [Ame03]: one is based on correspondence matching and another carries out explicit tracking by making use of position prediction or motion estimation. Useful mathematical tools for tracking include the Kalman filter, the Condensation algorithm, the dynamic Bayesian network, the Geodesic method, etc.

Tracking methods are divided into four major categories: region-based tracking, active-contour-based tracking, feature-based tracking and model-based tracking. It should be pointed out that this classification is not absolute in that algorithms from different categories can be integrated together.

4.4 Action recognition

After successfully tracking the moving objects from one frame to another in an image sequence, the problem of understanding objects' behaviors (action recognition) from image sequences follows naturally. Behavior understanding involves the analysis and recognition of motion patterns.

Understanding of behaviors may simply be thought as the classification of time varying feature data, i.e., matching an unknown test sequence with a group of labeled reference sequences representing typical behavior understanding is to learn the reference behavior sequences from training samples, and to devise both training and matching methods for coping effectively with small variations of the feature data within each class of motion patterns. This stage is out of the focus of this thesis. Otherwise the boosting technique presented in Chapter 6 could be used to learn the typical behavior and to detect unusual behaviors. The boosting technique developed

within LibAdaBoost is made generic and could be extended to cover the data model of behaviors and trajectories and other types of data.

4.5 Semantic description

Semantic description aims at describing object behaviors in a natural language suitable for nonspecialist operator of visual surveillance. Generally there are two major categories of behavior description methods: statistical models and formalized reasoning. This stage is out of the focus of this thesis.

4.6 Personal identification

The problem of *who is now entering the area under surveillance?* is of increasing importance for visual surveillance. Such personal identification can be treated as a special behavior understanding problem. For instance, human face and gait are now regarded as the main biometric features that can be used for people identification in visual surveillance systems [SEN98]. In recent years, great progress in face recognition has been achieved. The main steps in the face recognition for visual surveillance are face detection, face tracking, face feature detection and face recognition. Usually, these steps are studied separately. Therefore, developing an integrated face recognition system involving all of the above steps is critical for visual surveillance. This stage is out of the focus of this thesis.

4.7 Fusion of data from multiple cameras

Motion detection, tracking, behavior understanding, and personal identification discussed above can be realized by single camera-based visual surveillance systems. Multiple camera-based visual surveillance systems can be extremely helpful because the surveillance area is expanded and multiple view information can overcome occlusion. Tracking with a single camera easily generates ambiguity due to occlusion or depth. This ambiguity may be eliminated from another view. However, visual surveillance using multi cameras also brings problems such as camera installation (how to cover the entire scene with the minimum number of cameras), camera calibration, object matching, automated camera switching and data fusion.

Chapter 5

Moving object detection

Contents

Detection of moving objects and performing segmentation between background and moving objects is an important step in visual surveillance. Errors made at this abstraction level are very difficult to correct at higher levels. For example, when an object is not detected at the lowest level, it cannot be tracked and classified at the higher levels. The more accurate the segmentation at the lowest level, the better the object shape is known and consequently, the easier the (shape based) object recognition tasks become.

Frequently used techniques for moving object detection are background subtraction, statistical methods, temporal differencing and optical flow. Our system adopts a method based on combining different object detection algorithms of background modeling and subtraction, to achieve robust and accurate motion detection and to reduce false alarms. This approach is suitable for both color and infra-red sequences and based on the assumption of a stationary surveillance camera, which is the case in many surveillance applications.

Figure 5.1 shows an illustration of intermediate results within a moving object detection stage. Each new frame (Figure 5.1a) is used to update existing background image (Figure 5.1b). A threshold is applied to difference image between the current frame and the background image, and a binary image is produced (Figure 5.1c) which indicates the areas of activities. Finally, segmentation methods, applied on the binary image, can extract the moving objects (Figure 5.1d).

This chapter is organized as follows: section 5.1 provides a non exhaustive list of challenges that any object detection algorithm has to handle. Section 5.2 presents a generic template for an object detection system. In section 5.3 we describe our implementation of state of the art methods for adaptive background differencing. Shadows and light change detection elimination are described in section 5.4. The improvement of these methods using high level feedback is presented in section 5.5. To comparison of the different approaches on benchmarked datasets is presented in section 5.6.

5.1 Challenge of detection

There are several problems that a moving object detection algorithm must solve correctly. A non exhaustive list of problems would be the following [TKBM99]:

- Moved objects: A background object can be moved. These objects should not be considered part of the foreground forever after.

- Time of day: Gradual illumination changes alter the appearance of the background.

- Light switch: Sudden changes in illumination and other scene parameters alter the appearance of the background.

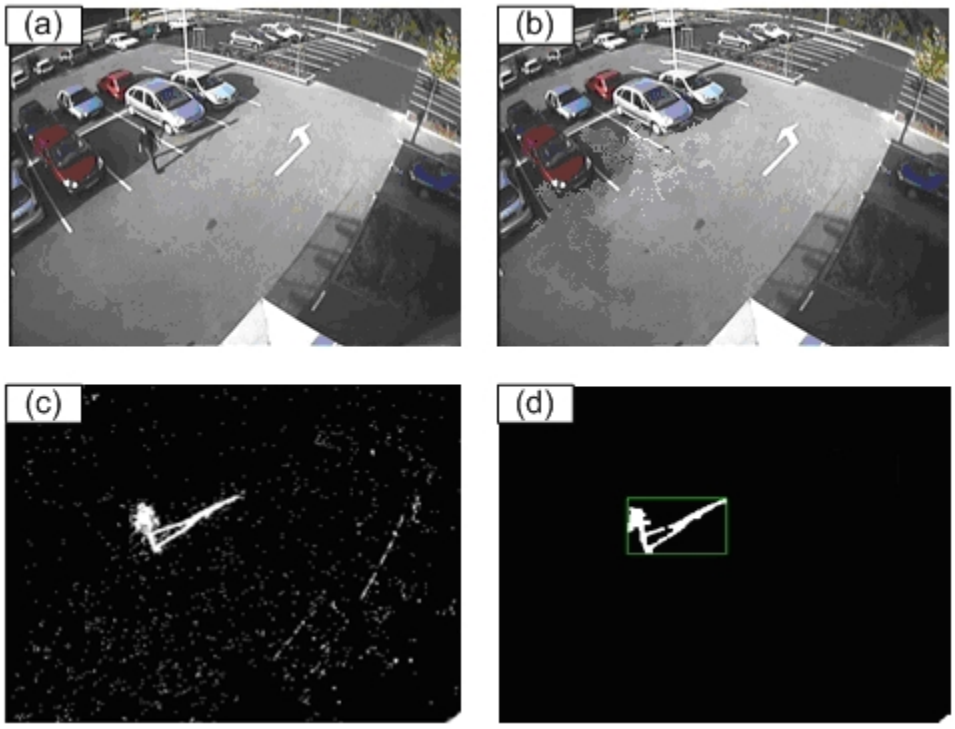


Figure 5.1: Object detection example: (a) input frame, (b) background image model, (c) foreground pixel-image, (d) detected objects

- Waving trees: Backgrounds can vacillate, requiring models which can represent disjoint sets of pixel values.
- Camouflage: A foreground object's pixel characteristics may be subsumed by the modeled background.
- Bootstrapping: A training period absent of foreground objects is not available in some environments.
- Foreground aperture: When a homogeneously colored object moves, change in the interior pixels cannot be detected. Thus, the entire object may not appear as foreground.
- Sleeping person: A foreground object that becomes motionless cannot be distinguished from a background object that moves and then becomes motionless.
- Waking person: When an object initially in the background moves, both it and the newly revealed parts of the background appear to change.
- Shadows: Foreground objects often cast shadows which appear different from the modeled background.

5.2 Object detection system diagram

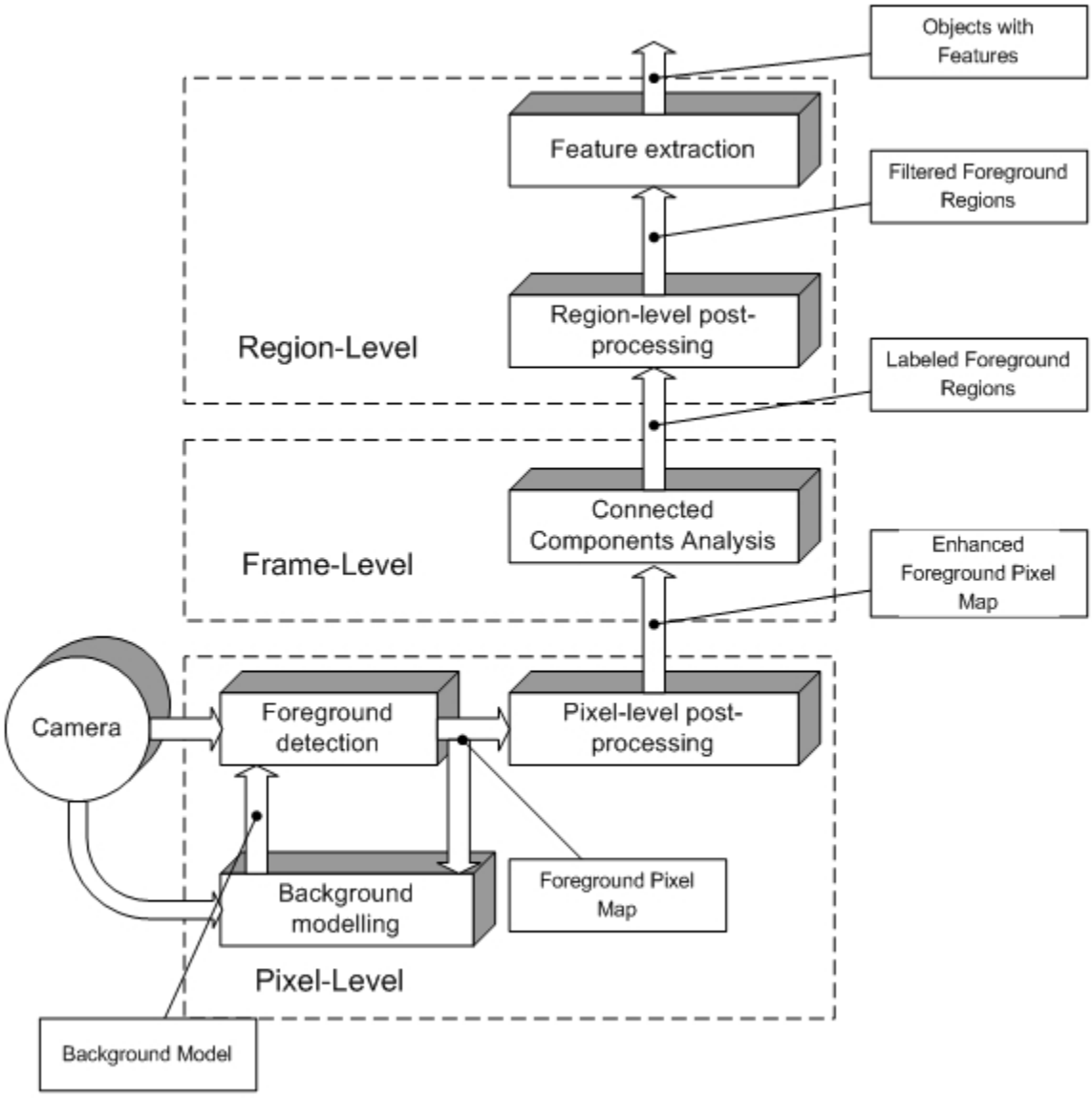


Figure 5.2: Object detection system diagram

Figure 5.2 shows a generic system diagram for our object detection method. The system starts by background scene initialization step. Next step is detecting the foreground pixels by using the background model and current frame from the video stream. This pixel-level detection method is dependent on the background model in use and it is used to update the background model to adapt to dynamic scene changes. The foreground detection step is followed by a pixel-level post-processing stage to perform some noise removal filters. Once the result enhanced foreground

pixel-map is ready, in the next step, connected components analysis is used to the labeling of the connected regions. The labeled regions are then filtered to eliminate small regions caused by environmental noise in the region-level post-processing step. More advanced filters are considered for merging overlapping isolated regions and will be detailed later. In the final step of the detection process, a number of object features are extracted for subsequent use by high level stages.

5.2.1 Foreground detection

A common approach to detecting the foreground regions is adaptive background differencing, where each video frame is compared against a reference, often called the *background image* or *background model*. The major steps in an adaptive background differencing algorithm are background modeling and foreground detection. Background modeling uses the new frame to calculate and update a background model. This background model provides a pixel-wise statistical representation of the entire background scene. Foreground detection then identifies pixels in the video frame that cannot be adequately explained by the background model, and outputs them as a binary candidate foreground mask.

Several methods for adaptive background differencing have been proposed in the recent literature. All of these methods try to effectively estimate the background model from the temporal sequence of the frames. However, there are a wide variety of techniques and both the expert and the newcomer to this area can be confused about the benefits and limitations of each method. In our video surveillance system we have selected five algorithms for modeling the background :

1. Basic Background Segmentation (BBS).
2. W4 method (W4).
3. Single Gaussian Model (SGM).
4. Mixture Gaussian Model (MGM).
5. Lehigh omni-directional Tracking System (LOTS).

The first method is a basic background subtraction algorithm (BBS) [HA05]. This is the simplest algorithm and it provides a lower benchmark for the other algorithms which are more complex but based on the same principle.

The second algorithm is denoted as W4 and operates on gray scale images. Three parameters are learned for each pixel to model the background: minimum intensity, maximum intensity and maximum absolute difference in consecutive frames. This algorithm incorporates the noise variations into the background model.

The third method is used in Pfinder [WATA97] denoted here as SGM (Single Gaussian Model). This method assumes that each pixel is a realization of a random variable with a Gaussian distribution. The first and second order statistics of this distribution are independently estimated for each pixel.

The fourth method is an adaptive mixture of multiple Gaussian (MGM) as proposed by Stauffer and Grimson in [CG00]. Every pixel of the background is modeled using a mixture of Gaussian. The weights of the mixture and the parameters of the Gaussian are adapted with respect to the current frame. This method has the advantage that multi modal backgrounds (such as moving trees) can be modeled. Among the tested techniques, this is the one with the most complex background model.

The fifth approach (LOTS) proposed by Boult in [BMGE01] is an efficient method designed for military applications that presumes a two background model. In addition, the approach uses high and low per-pixel thresholds. The method adapts the background by incorporating the current image with a small weight. At the end of each cycle, pixels are classified as false detection, missed detection and correct detection. The original point of this algorithm is that the per-pixel thresholds are updated as a function of the classification.

5.2.2 Pixel level post-processing (Noise removal)

The outputs of foreground region detection algorithms generally contain noise and therefore are not appropriate for further processing without special post-processing. There are various factors that cause the noise in foreground detection, such as:

1. Camera noise: This is the noise caused by the cameras image acquisition components. The intensity of a pixel that corresponds to an edge between two different colored objects in the scene may be set to one of the object color in one frame and to the other object color in the next frame.

2. Reflectance noise: When a source of light, for instance sun, moves it makes some parts in the background scene to reflect light. This phenomenon makes the foreground detection algorithms fail and detect reflectance as foreground regions.

3. Background colored object noise: Some parts of the objects may have the same color as the reference background behind them. This resemblance causes some of the algorithms to detect the corresponding pixels as non-foreground and objects to be segmented inaccurately.

4. Shadows and sudden illumination change: Shadows cast on objects are detected as foreground by most of the detection algorithms. Also, sudden illumination changes (e.g. turning on lights in a monitored room) makes the algorithms fail to detect actual foreground objects accurately.

Morphological operations, erosion and dilation [Hei96], are applied to the foreground pixel map in order to remove noise that is caused by the first three of the items listed above. Our aim in applying these operations is removing noisy foreground pixels that do not correspond to actual foreground regions and to remove the noisy background pixels near and inside object regions that are actually foreground pixels. Erosion,

as its name implies, erodes one-unit thick boundary pixels of foreground regions. Dilation is the reverse of erosion and expands the foreground region boundaries with one-unit thick pixels. The subtle point in applying these morphological filters is deciding on the order and amount of these operations. The order of these operations affects the quality and the amount affects both the quality and the computational complexity of noise removal.

Removal of shadow regions and detecting and adapting to sudden illumination changes require more advanced methods which are explained in section 5.4.

5.2.3 Detecting connected components

After detecting foreground regions and applying post-processing operations to remove noise and shadow regions, the filtered foreground pixels are grouped into connected regions (blobs) and labeled by using a two-level connected component labeling algorithm presented in [Hei96]. After finding individual blobs that correspond to objects, the bounding boxes of these regions are calculated.

5.2.4 Region level post-processing

Even after removing pixel-level noise, some artificial small regions remain due to inaccurate object segmentation. In order to eliminate this type of regions, the average region size (γ) in terms of pixels is calculated for each frame and regions that have smaller sizes than a fraction (α) of the average region size ($Size(region) < \alpha * \gamma$) are deleted from the foreground pixel map.

Also, due to segmentation errors, some parts of the objects are found as disconnected from the main body. In order to correct this effect, the bounding boxes of regions that are close to each other are merged together and the region labels are adjusted.

5.2.5 Extracting object features

Once we have segmented regions we extract features of the corresponding objects from the current image. These features are size (S), center of mass (C_m), color histogram (H_c) and silhouette contour of the objects blob. Calculating the size of the object is trivial and we just count the number of foreground pixels that are contained in the bounding box of the object. In order to calculate the center of mass point, $C_m = (x_{C_m}, y_{C_m})$, of an object O, we use the following equation 5.1:

$$x_{C_m} = \frac{\sum x_i}{n}, y_{C_m} = \frac{\sum y_i}{n} \tag{5.1}$$

where n is the number of pixels in O. The color histogram, H_c is calculated over monochrome intensity values of object pixels in current image. In order to reduce computational complexity of operations that use H_c, the color values are quantized. Let N be the number of bins in the histogram, then every bin covers $\frac{255}{N}$ color values.

The color histogram is calculated by iterating over pixels of O and incrementing the stored value of the corresponding color bin in the histogram, H_c. So for an object O the color histogram is updated as follows:

$$H_c[\frac{c_i}{N}] = H_c[\frac{c_i}{N}] + 1, \forall c_i \in O \tag{5.2}$$

where c_i represents the color value of i^th pixel. In the next step the color histogram is normalized to enable appropriate comparison with other histograms in later steps. The normalized histogram $\hat{H}_c$ is calculated as follows:

$$\hat{H}_c[i] = \frac{H_c[i]}{\sum H_c[i]} \tag{5.3}$$

5.3 Adaptive background differencing

This section describes object detection algorithms based on adaptive background differencing technique used in this work: BBS, W4, SGM, MGM and LOTS. The BBS, SGM, MGM algorithms use color images while W4 and LOTS use gray scale images.

5.3.1 Basic Background Subtraction (BBS)

There are different approaches to this basic scheme of background subtraction in terms of foreground region detection, background maintenance and post processing. This method detects targets by computing the difference between the current frame and a background image for each color channel RGB. The implementation can be summarized as follows:

- **Background initialization and threshold:** motion detection starts by computing a pixel based absolute difference between each incoming frame I_t and an adaptive background frame B_t. The pixels are assumed to contain motion if the absolute difference exceeds a predefined threshold level τ. As a result, a binary image is formed if (5.4) is satisfied, where active pixels are labeled with 1 and non-active ones with 0.

  $$|I_t(\phi) - B_t(\phi)| > \tau \tag{5.4}$$

  where τ is a predefined threshold. The operation in (5.4) is performed for all image pixels ϕ.

- **Filtering:** the resulting thresholded image usually contains a significant amount of noise. Then a morphological filtering step is applied with a 3x3 mask (dilation and erosion) that eliminates isolated pixels.

- **Background adaptation:** to take into account slow illumination changes which is necessary to ensure longterm tracking, the background image is subsequently updated by:

$$B_i^{t+1}(\phi) = \alpha I_i^t(\phi) + (1-\alpha)B_i^t(\phi) \tag{5.5}$$

where α is the learning rate. In our experiments we use $\alpha = 0.15$ and the threshold $\tau = 0.2$. These parameters stay constant during the experiment.

The following figure shows the results

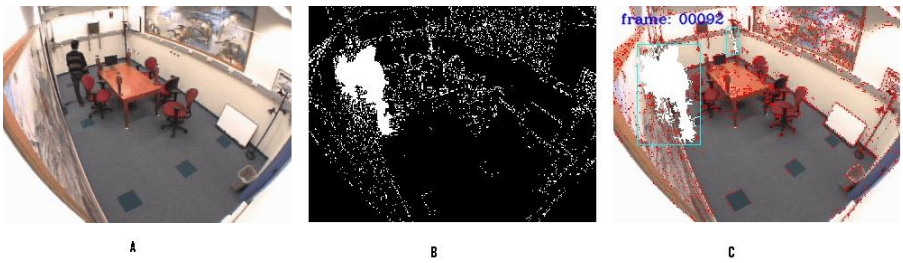


Figure 5.3: The BBS results (a) current frame (b)(c) Foreground detection

5.3.2 W4 method

This algorithm was proposed by Haritaoglu in [CG00]. W4 has been designed to work with only monochromatic stationary video sources, either visible or infrared. It is designed for outdoor detection tasks, and particularly for night-time or other low light level situations.

- **Background scene modeling:** W4 obtains the background model even if there are moving foreground objects in the field of view, such as walking people moving cars, etc. The background scene is modeled by representing each pixel by three values; minimum intensity(m), maximum intensity (M), and the maximum intensity difference (D) between consecutive frames. In our implementation we selected an array V containing N consecutive images, $V^i(\phi)$ is the intensity of a pixel location ϕ in the i^{th} image of V. The initial background model for a pixel location $[m(\phi), M(\phi), D(\phi)]$, is obtained as follows:

$$\begin{bmatrix} m(\phi) \\ M(\phi) \\ D(\phi) \end{bmatrix} = \begin{bmatrix} minV(\phi) \\ maxV(\phi) \\ max|V^i(\phi) - V^{i-1}(\phi)| \end{bmatrix} \tag{5.6}$$

- **Classification:** each pixel is classified as background or foreground according to (5.7). Giving the values of Min, Max and D, a pixel $I(\phi)$ is considered as foreground pixel if :

$$|m(\phi) - I(\phi)| > D(\phi) or |M(\phi) - I(\phi)| > D(\phi) \tag{5.7}$$

- **Filtering:** after classification the resulting image usually contains a significant amount of noise. A region based noise cleaning algorithm is applied that is composed of an erosion operation followed by connected component analysis that allows to remove regions with less than 50 pixels. The morphological operations dilatation and erosion are now applied.

Figure 5.4: The w4 results (a) current frame (b)(c) Foreground detection

5.3.3 Single Gaussian Model (SGM)

The Single Gaussian Model algorithm is proposed by Wren in [WWT03]. In this section we describe the different steps to implement this method.

- **Modeling the background:** we model the background as a texture surface; each point on the texture surface is associated with a mean color value and a distribution about that mean. The color distribution of each pixel is modeled with the Gaussian described by a full covariance matrix. The intensity and color of each pixel are represented by a vector $[Y, U, V]^T$. We define $\mu(\phi)$ to be the mean $[Y, U, V]$ of a point on the texture surface, and $U(\phi)$ to be the covariance of that point's distribution.

- **Update models:** we update the mean and covariance of each pixel ϕ as follows:

$$\mu(\phi) = (1-\alpha)\mu^{t-1}(\phi) + \alpha I^t(\phi), \tag{5.8}$$
$$U^t(\phi) = (1-\alpha)U^{t-1}(\phi) + \alpha\nu(\phi)\nu(\phi)^T \tag{5.9}$$

  Where $I^t(\phi)$ is the pixel of the current frame in YUV color space, α is the learning rate; in the experiments we have used $\alpha = 0.005$ and $\nu(\phi) = I^t(\phi) - \mu^t(\phi)$.

- **Detection:** for each image pixel $I^t(\phi)$ we must measure the likelihood that it is a member of each of the blob models and the scene model. We compute for each pixel at the position ϕ the log likelihood $L(\phi)$ of the difference $\nu(\phi)$. This value gives rise to a classification of individual pixels as background or foreground.

$$L(\phi) = -\frac{1}{2}\nu(\phi)^T(U^t)^{-1} - \frac{1}{2}\ln|U^t| - \frac{3}{2}\ln(2\pi). \tag{5.10}$$

A pixel ϕ is classified as foreground if $L(\phi) > \tau$ otherwise it is classified as background. Where τ is a threshold, in the experiment we consider $\tau = -300$. We have obtained the following results (see on Figure 5.5)

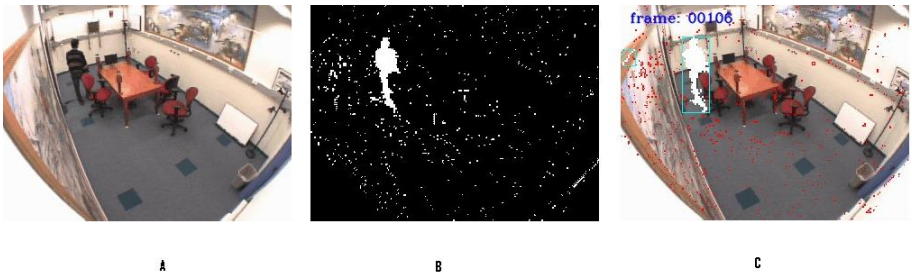


Figure 5.5: The SGM results (a) current frame (b)(c) Foreground detection

5.3.4 Mixture Gaussian Model (MGM)

In this section we present the implementation of the original version of the adaptive Mixture of multiple Gaussians Model for background modeling. This model was proposed by Stauffer and Grimson in [HA05]. It can robustly deal with lighting changes, repetitive motions, clutter, introducing or removing objects from the scene and slowly moving objects. Due to its promising features, this model has been a popular choice for modeling complex and dynamic backgrounds. The steps of the algorithm can be summarised as:

- **Background modeling:** in this model, the value of an individual pixel (e.g. scalars for gray values or vectors for color images) over time is considered as a *pixel process* and the recent history of each pixel, $\{X1, ..., Xt\}$, is modeled by a mixture of K Gaussian distributions. The probability of observing current pixel value then becomes:

$$P(X_t) = \sum_{i=1}^{k} w_{i,t} * \eta(X_t, \mu_{i,t}, \sigma^2_{i,t}) \tag{5.11}$$

where k is the number of distributions, $w_{i,t}$ is an estimate of the weight (what portion of the data is accounted for this Gaussian) of the i^{th} Gaussian $G_{i,t}$ in the mixture at time t, $\mu_{i,t}$ is the mean value of $G_{i,t}$ and $\sigma^2_{i,t}$ is the covariance matrix of $G_{i,t}$ and η is a Gaussian probability density function:

$$\eta(X_t, \mu, \sigma^2) = \frac{1}{(2\pi)^{\frac{n}{2}}|\sigma^2|^{\frac{1}{2}}} e^{-\frac{1}{2}(X_t-\mu_t)^T(\sigma^2)^{-1}(X_t-\mu_t)}. \tag{5.12}$$

Decision on K depends on the available memory and computational power. Also, the covariance matrix is assumed to be of the following form for computational efficiency

$$\sigma^2_{k,t} = \alpha^2_k I \tag{5.13}$$

which assumes that red, green and blue color components are independent and have the same variance.

- **Update equations:** the procedure for detecting foreground pixels is as follows: when the system starts, the K Gaussian distributions for a pixel are initialized with predefined mean, high variance and low prior weight. When a new pixel is observed in the image sequence, to determine its type, its RGB vector is checked against the K Gaussians, until a match is found. A match is defined if the pixel value is within $\gamma = 2.5$ standard deviations of a distribution. Next, the prior weights of the K distributions at time t, $w_{k,t}$, are updated as follows:

$$w_{k,t} = (1 - \alpha)w_{k,t-1} + \alpha(M_{k,t}) \tag{5.14}$$

Where α is the learning rate and $M_{k,t}$ is 1 for the model which matched the pixel and 0 for the remaining models. After this approximation the weights are renormalised. The changing rate in the model is defined by $\frac{1}{\alpha}$. The parameters μ and σ^2 for unmatched distribution are updated as follows:

$$\mu_t = (1 - \alpha)\mu_{t-1} + \rho X_t \tag{5.15}$$
$$\sigma_t^2 = (1 - \rho)\sigma_{t-1}^2 + \rho(X_t - \mu)^T \tag{5.16}$$
$$\rho = \alpha\nu(X_t|\mu_k, \sigma_k) \tag{5.17}$$

If no match is found for the new observed pixel, the Gaussian distribution with the least probability is replaced with a new distribution with the current pixel value as its mean value, an initially high variance and low prior weight.

- **Classification:** in order to classify (foreground or background) a new pixel, the K Gaussian distributions are sorted by the value of w/σ. This ordered list of distributions reflects the most probable backgrounds from top to bottom since by 5.14, background pixel processes make the corresponding Gaussian distribution have larger prior weight and less variance. Then the first B distributions are defined as background.

$$B = \arg min_b(\sum_{k=1}^{b} w_k > T) \tag{5.18}$$

T is the minimum portion of the pixel data that should be accounted for by the background. If a small value is chosen for T, the background is generally unimodal. Figure 5.6 shows the results.

5.3.5 Lehigh Omni-directional Tracking System (LOTS):

The target detector proposed in [WATA97] operates on gray scale images. It uses two background images and two per-pixel thresholds. The steps of the algorithm can be summarized as:

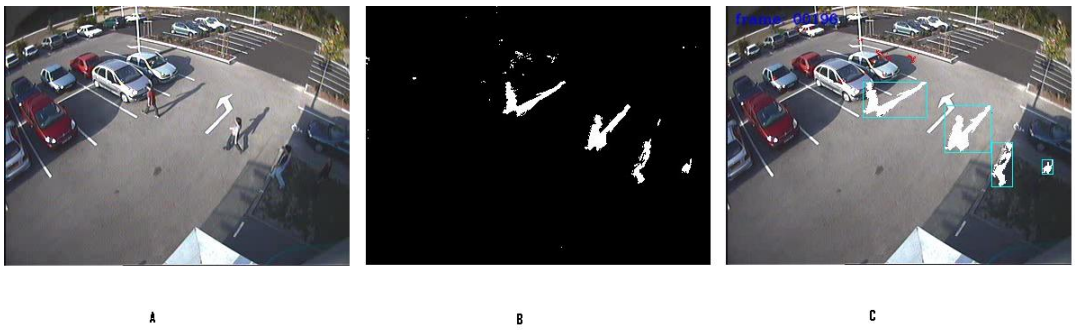


Figure 5.6: The MGM results (a) Current frame (b) (c) Foreground detection

- **Background and threshold initialization:** the first step of the algorithm, performed only once, assumes that during a period of time there are no targets in the image. In this ideal scenario, the two backgrounds are computed easily. The image sequences used in our experiment do not have target-free images, so B_1 is initialized as the mean of 20 consecutive frames and B_2 is given by :

$$B_2 = B_1 + \mu \tag{5.19}$$

  Where μ is an additive Gaussian noise with $\eta(\mu = 10, \sigma = 20)$ The per-pixel threshold T_L, is then initialized to the difference between the two backgrounds:

$$T_L(\phi) = |B_1(\phi) - B_2(\phi)| + \nu(\phi) \tag{5.20}$$

  Where ν represents noise with an uniform distribution in [1,10], and ϕ an image pixel. Then the higher threshold T_H is computed by:

$$T_H(\phi) = T_L(\phi) + V \tag{5.21}$$

  Where V is the sensitivity of the algorithm, we have chosen $V = 40$ in our implementation.

- **Detection:** we create D, which contains the minimum of the differences between the new image I, and the backgrounds B_1 and B_2:

$$D(\phi) = min_j |I_j(\phi) - B_j(\phi)|, j = 1, 2. \tag{5.22}$$

  D is then compared with the threshold images. Two binary images D_L and D_H are created. The active pixels of D_L and D_H are the pixels of D that are higher than the thresholds T_L and T_H respectively.

- **Quasi connected components analysis (QCC):** QCC computes for each thresholded image D_L and D_H images Ds_L and Ds_H with 16 times smaller resolution. Each element in these reduced images, Ds_L and Ds_H, has the number of active pixels in a 4x4 block of D_L and D_H respectively. Both images are then merged into a single image that labels pixels as detected, missed or insertions. This process labels every pixel, and also deals with targets that are not completely connected, considering them as only one region.

- **Labeling and cleaning:** a 4-neighbor connected components analyzer is then applied to this image (Ds_L and Ds_H) and, the regions with less than Λ pixels are eliminated. The remaining regions are considered as detected. The detected pixels are the ones from D_L that correspond to detected regions. The insertions are the active pixels in D_L, but do not correspond to detected regions, and the missed pixels are the inactive pixels in D_L.

- **Background and threshold adaptation:** The backgrounds are updated as follows:

$$B_i^{t+1}(\phi) = \begin{cases} (1-\alpha_2)B_i^t(\phi) + \alpha_2 I^t(\phi) & \phi \in T^t \\ (1-\alpha_1)B_i^t(\phi) + \alpha_1 I^t(\phi) & \phi \in N^t \end{cases} \tag{5.23}$$

  Where $\phi \in T^t$ represents a pixel that is labeled as detected and $\phi \in N^t$ represents a missed pixel. Generally α_2 is smaller than α_1, In our experiment we have chosen $\alpha_1 = 0.000306$ and $\alpha_2 = \frac{\alpha_1}{4}$, and only the background corresponding to the smaller difference D is updated. Using the pixel labels, thresholds are updated as follows:

$$T_L^{t+1}(\phi) = \begin{cases} T_L^t(\phi) + 10 & \phi \in \text{ insertion} \\ T_L^t(\phi) - 1 & \phi \in \text{ missed} \\ T_L^t(\phi) & \phi \in \text{ detected} \end{cases} \tag{5.24}$$

  We have obtained the following results (see on Figure 5.7).

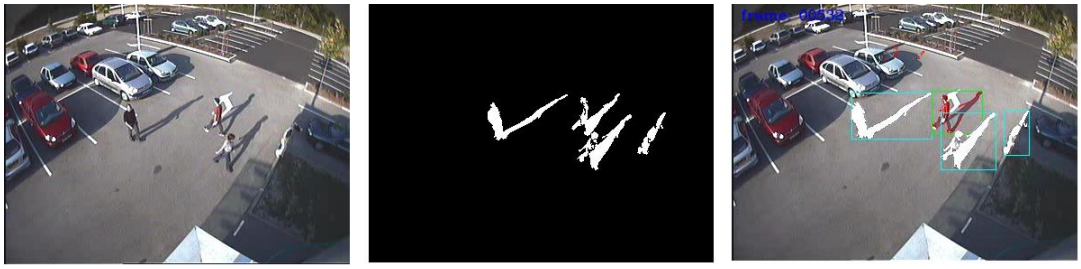


Figure 5.7: The LOTS results (a) Current frame (b)(c) Foreground detection

5.4 Shadow and light change detection

The algorithms described above for motion detection perform well on indoor and outdoor environments and have been used for real-time surveillance for years. However, most of these algorithms are susceptible to both local (e.g. shadows and highlights) and global illumination changes (e.g. sun being covered/uncovered by clouds). Shadows cause the motion detection methods fail in segmenting only the moving objects

and make the upper levels such as object classification to perform inaccurately. Many existing shadow detection algorithms mostly use either chromaticity or a stereo information to cope with shadows and sudden light changes. In our video surveillance system we have implemented two methods based on chromaticity: Horprasert [TDD99] and Chen [CL04].

Shadows are due to the occlusion of the light source by an object in the scene (Figure 5.8). The part of an object that is not illuminated is called self-shadow, while the area projected on the scene by the object is called cast shadow and is further classified into umbra and penumbra. The umbra corresponds to the area where the direct light is totally blocked by the object, whereas in the penumbra area it is partially blocked. The umbra is easier to be seen and detected, but is more likely to be misclassified as moving object. If the object is moving, the cast shadow is more properly called moving cast shadow [JRO99], otherwise is called still shadow. Human visual system seems to utilize the following observations for reliable shadow detection:

1. Shadow is dark but does not change significantly neither the color nor the texture of the background covered.

2. Shadow is always associated with the object that casts it and to the behavior of that object.

3. Shadow shape is the projection of the object shape on the background. For an extended light source,the projection is unlikely to be perspective.

4. Both the position and strength of the light source are known.

5. Shadow boundaries tend to change direction according to the geometry of the surfaces on which they are cast.

Figure 5.8: Illumination of an object surface

5.4.1 Methods and implementation

Shadow detection processes can be classified in two ways. One, they are either deterministic or statistical, and two, they are either parametric or non-parametric. This

particular approach for shadow detection is statistical and non-parametric which implies that it uses probabilistic functions to describe the class membership and parameter selection is not a critical issue. In the current work, to cope with shadows, we selected implemented and evaluated two methods: a statistical non-parametric approach introduced by Horprasert et al. in [TDD99] and a deterministic non-model based approach proposed by Chen et al. [CL04]. Our objective is to complete our video surveillance prototype with shadow removal cores with known performance and limitations.

Statistical non-parametric approach

A good understanding of this approach requires the introduction of some mathematical tools for image processing:

- **Computational color model:** based on the fact that humans tend to assign a constant color to an object even under changing illumination over time and space (color constancy), the color model used here separates the brightness from the chromaticity component. Figure 5.9 illustrates the proposed color model in

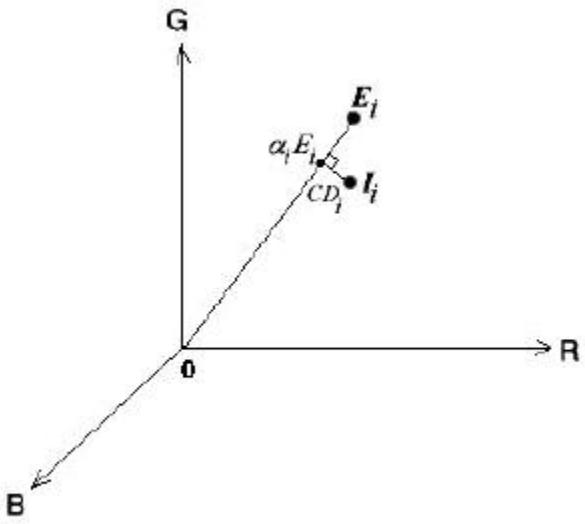


Figure 5.9: Proposed color model in three dimensional RGB color space

three-dimensional RGB color space. Consider a pixel, i, in the image let $E_i = [E_R(i), E_G(i), E_B(i)]$ represent the pixel's expected RGB color in the background image. Next, let $I_i = [I_R(i), I_G(i), I_B(i)]$ denotes the pixel's RGB color value in a current image that we want to subtract from the background. The line OE_i is called expected chromaticity line whereas OI_i is the observed color value. Basically, we want to measure the distortion of I_i from E_i. The proposed color model separates the brightness from the chromaticity component. We do this

by decomposing the distortion measurement into two components, brightness distortion and chromaticity distortion, which will be described below.

- **Brightness distortion:** the brightness distortion (α) is a scalar value that brings the observed color close to the expected chromaticity line. It is obtained by minimizing (5.25):

$$\phi(\alpha_i) = (I_i - \alpha_i E_i)^2 \tag{5.25}$$

- **Color distortion:** color distortion is defined as the orthogonal distance between the observed color and the expected chromaticity line. The color distortion of a pixel i is given by (5.26).

$$CD_i = ||I_i - \alpha_i E_i|| \tag{5.26}$$

- **Color characteristics:** the CCD sensors linearly transform infinite-dimensional spectral color space to a three-dimensional RGB color space via red, green, and blue color filters. There are some characteristics of the output image, influenced by typical CCD cameras, that we should account for in designing the algorithm, as follows:

  – color variation: the RGB color value for a given pixel varies over a period of time due to camera noise and illumination fluctuation by light sources.

  – band unbalancing: cameras typically have different sensitivities to different colors. Thus, in order to make the balance weights on the three color bands (R,G,B), the pixel values need to be rescaled or normalized by weight values. Here, the pixel color is normalized by its standard deviation (s_i) which is given by (5.27)

$$s_i = [\sigma_R(i), \sigma_G(i), \sigma_B(i)] \tag{5.27}$$

  where $\sigma_R(i)$, $\sigma_G(i)$, and $\sigma_B(i)$ are the standard deviation of the i_th pixel's red, green, blue values computed over N frames of the background frames.

  – clipping: since the sensors have limited dynamic range of responsiveness, this restricts the varieties of color into a RGB color cube, which is formed by red, green, and blue primary colors as orthogonal axes. On 24-bit images, the gamut of color distribution resides within the cube range from [0, 0, 0] to [255, 255, 255]. Color outside the cube (negative or greater than 255 color) cannot be represented. As a result, the pixel value is clipped in order to lie entirely inside the cube.

The algorithm is based on pixel modeling and background subtraction and its flowchart is shown in Figure 5.10. Typically, the algorithm consists of three basic steps:

1. **Background modeling:** the background is modeled statistically on a pixel by pixel basis. A pixel is modeled by a 4-tuple $< E_i, s_i, a_i, b_i >$ where E_i is

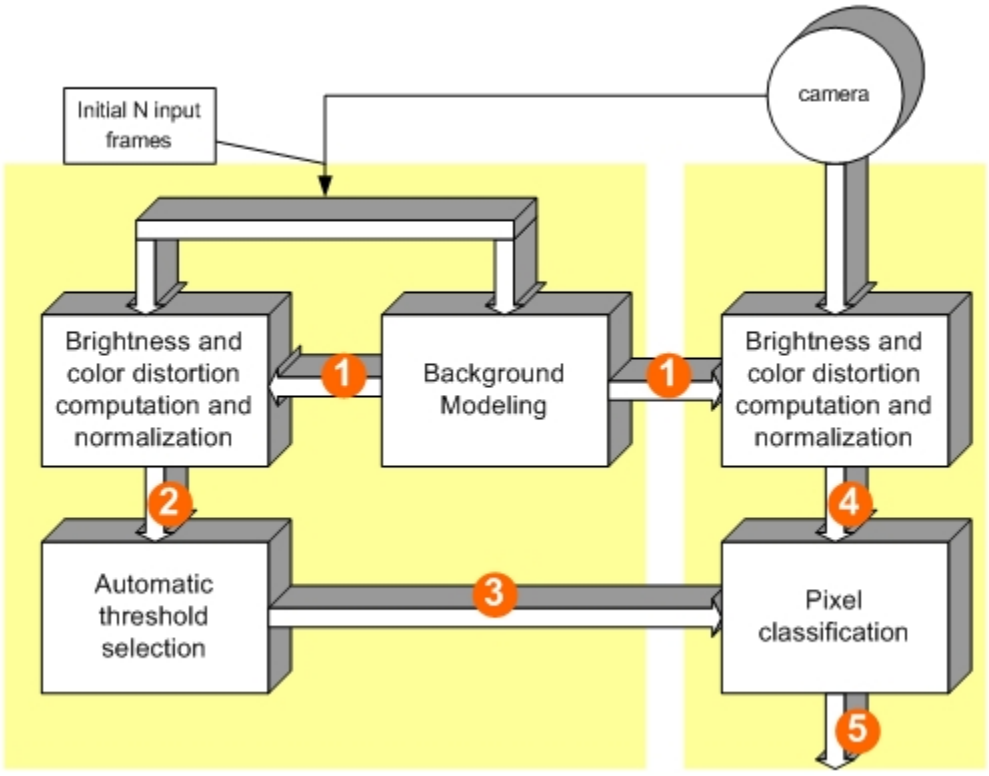


Figure 5.10: Flowchart of the Statistical Non Parametric approach. The first N frames are used to compute, for each pixel, the means and the variances of each color channel, E_i and S_i respectively (1). Then, the distortion of the brightness α_i and the distortion of the chrominance CD_i of the difference between expected color of a pixel and its value in the current image are computed and normalized (2,4). Finally, each pixel is classified into four classes $C(i)$ (5) using a decision rule based on thresholds τ_{CD}, $\tau_{\alpha_{lo}}$, $\tau\alpha_1$ and τ_{α_2} automatically computed (3).

the expected color value, s_i is the standard deviation of color value , a_i is the variation of the brightness distortion, and b_i is the variation of the chromaticity distortion of the i^{th} pixel. The background image and some other associated parameters are calculated over a number of static background frames. The expected color value of pixel i is given by (5.28)

$$E_i = [\mu_R(i), \mu_G(i), \mu_B(i)] \tag{5.28}$$

where $\mu_R(i)$, $\mu_G(i)$, and $\mu_B(i)$ are the arithmetic means of the i_{th} pixel's red, green, blue values computed over N background frames. Since the color bands have to be balanced by rescaling color values by pixel variation factors, the formula for calculating brightness distortion and chromaticity distortion can be written as:

$$\alpha_i = \frac{\frac{I_R(i)\mu_R(i)}{{\sigma_R}^2(i)} + \frac{I_G(i)\mu_G(i)}{{\sigma_G}^2(i)} + \frac{I_B(i)\mu_B(i)}{{\sigma_B}^2(i)}}{[\frac{\mu_R(i)}{\sigma_R(i)}]^2 + [\frac{\mu_G(i)}{\sigma_G(i)}]^2 + [\frac{\mu_B(i)}{\sigma_B(i)}]^2} \tag{5.29}$$

$$CD_i = \sqrt{(\frac{I_R(i) - \alpha_i\mu_R(i)}{\sigma_R(i)})^2 + (\frac{I_G(i) - \alpha_i\mu_G(i)}{\sigma_G(i)})^2 + (\frac{I_B(i) - \alpha_i\mu_B(i)}{\sigma_B(i)})^2} \tag{5.30}$$

Since different pixels yield different distributions of brightness and chromaticity distortions, these variations are embedded in the background model as a_i and b_i respectively which are used as normalization factors.

The variation in the brightness distortion of the i^{th} pixel is given by a_i, which is given by:

$$a_i = RMS(\alpha_i) = \sqrt{\frac{\sum_{i=0}^{N}(\alpha_i - 1)^2}{N}} \tag{5.31}$$

The variation in the chromaticity distortion of the i^{th} pixel is given by b_i which is defined as:

$$b_i = RMS(CD_i) = \sqrt{\frac{\sum_{i=0}^{N}(CD_i)^2}{N}} \tag{5.32}$$

In order to use a single threshold for all pixels, we need to rescale i and CD_i. Therefore, let

$$\widehat{\alpha_i} = \frac{\alpha_i - i - 1}{a_i} \tag{5.33}$$

and

$$\widehat{CD_i} = \frac{CD_i}{b_i} \tag{5.34}$$

be normalized brightness distortion and normalized chromaticity distortion respectively.

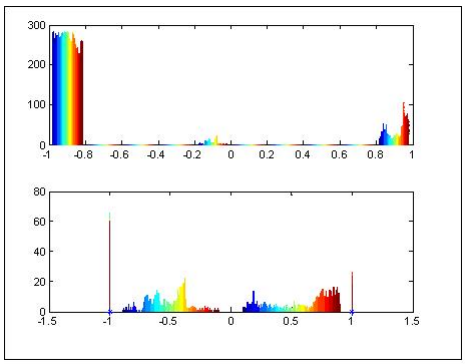


Figure 5.11: Histogram of Normalized alpha

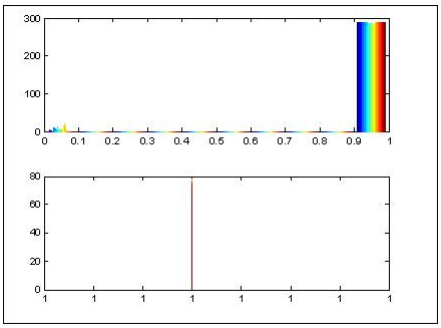


Figure 5.12: Histogram of Normalized CD

2. **Threshold selection:** a statistical learning procedure is used to automatically determine the appropriate thresholds. The histograms of both $\widehat{\alpha_i}$ and the $\widehat{CD_i}$ are built and the thresholds are computed by fixing a certain detection rate. τ_{CD} is a threshold value used to determine the similarity in chromaticity between the background image and the current observed image and τ_{alpha_1} and τ_{alpha_2} are selected threshold values used to determine the similarities in brightness. $\tau_{\alpha_{lo}}$ is described in the classification step.

   According to histograms of normalized α (Figure 5.11) and CD (Figure 5.12) the values selected for the various thresholds to be utilized for pixel classification are as follows (see on Table 5.1).

Threshold	Value
τ_{CD}	100
$\tau_{\alpha_{lo}}$	-50
τ_{α_1}	15
τ_{α_2}	-15

Table 5.1: Shadow detection thresholds learning result

3. **Pixel classification:** chromaticity and brightness distortion components are calculated based on the difference between the background image and the current image. Suitable threshold values are applied to these components to determine the classification of each pixel i according to the following definitions:

   - Background (B): if it has both brightness and chromaticity similar to those of the same pixel in the background image.
   - Shaded background or Shadow (S): if it has similar chromaticity but lower brightness than those of the same pixel in the background image. This is based on the notion of the shadow as a semi-transparent region in the

image, which retains a representation of the underlying surface pattern, texture or color value.

- Highlighted background (H): if it has similar chromaticity but higher brightness than the background image.
- Foreground (F): if the pixel has chromaticity different from the expected values in the background image.

Based on these definitions, a pixel is classified into one of the four categories $C(i) \in \{F, B, S, H\}$ by the following decision procedure.

$$C(i) = \begin{cases} F : \widehat{CD_i} > \tau_{CD}, \text{ else} \\ B : \widehat{\alpha}_i < \tau_{\alpha_1} \text{ and } \widehat{\alpha}_i > \tau_{\alpha_2}, \text{ else} \\ S : \widehat{\alpha}_i < 0, \text{ else} \\ H : otherwize \end{cases} \tag{5.35}$$

A problem encountered here is that if a moving object in a current image contains very low RGB values, this dark pixel will always be misclassified as a shadow. This is because the color point of the dark pixel is close to the origin in RGB space and the fact that all chromaticity lines in RGB space meet at the origin, which causes the color point to be considered close or similar to any chromaticity line. To avoid this problem, we introduce a lower bound for the normalized brightness distortion ($\tau\alpha lo$). Then, the decision procedure becomes:

$$C(i) = \begin{cases} F : \widehat{CD_i} > \tau_{CD} or \widehat{\alpha}_i < \tau\alpha_{lo}, \text{ else} \\ B : \widehat{\alpha}_i < \tau_{\alpha_1} \text{ and } \widehat{\alpha}_i > \tau_{\alpha_2}, \text{ else} \\ S : \widehat{\alpha}_i < 0, \text{ else} \\ H : otherwize \end{cases} \tag{5.36}$$

Deterministic non-model based approach

A shadow cast on a background does not change its hue significantly, but mostly lower the saturation of the points. We observe that the relationship between pixels when illuminated and the same pixels under shadows is roughly linear. Based on those observations, we exploit the brightness distortion and chrominance distortion of the difference between the expected color of a pixel and its value in the current image; otherwise the saturation component information and the ratios of brightness between the current image and the reference image is employed as well to determine the shadowed background pixels. This method is close to the method proposed by Chen et al [CL04] which presents a background model initiation and maintenance algorithm by exploiting HSV color information.

This method differs from the previously described statistical non-parametric approach in Section 5.4.1 by the way it handles its background model. This approach does not provide a special procedure for updating the initial background model but

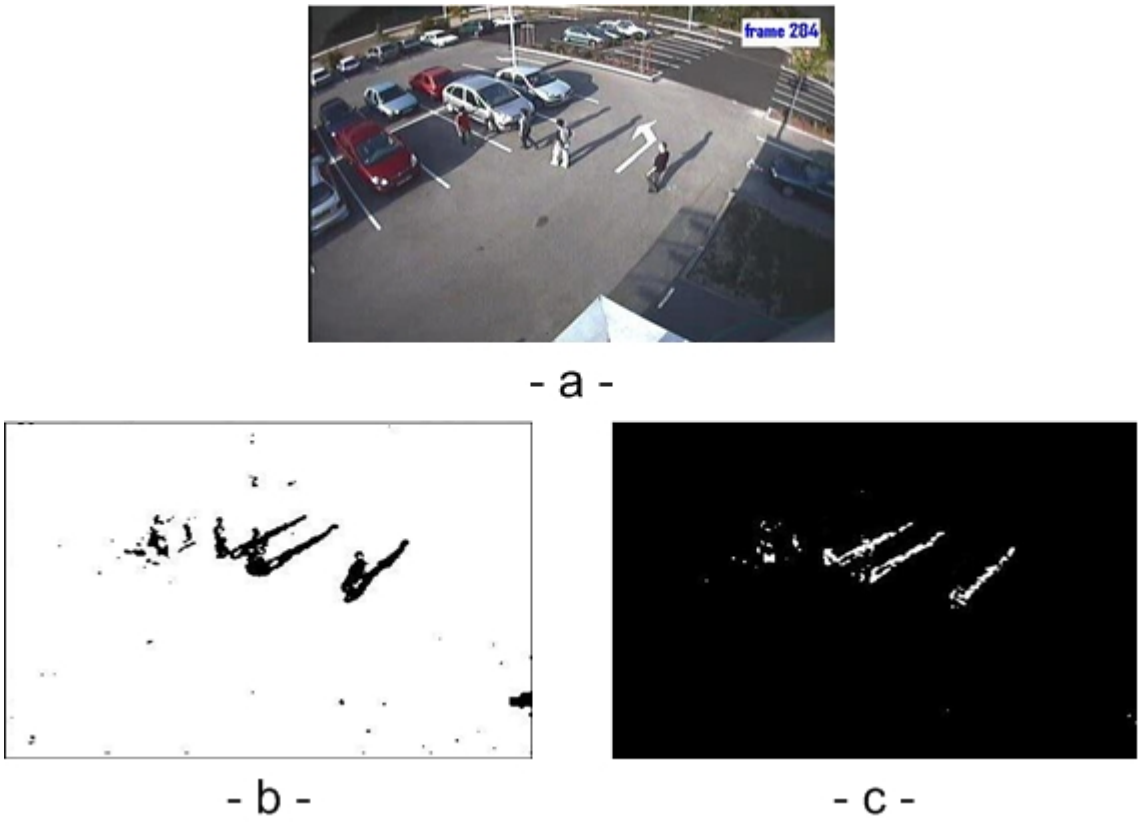


Figure 5.13: Result of Statistical non-parametric approach:(a) current frame, (b) without shadow detection result (c) shadows

it uses instead, a similar procedure to create the background model periodically each N frames. In other words, it provides a newest background model periodically. Figure 5.14 shows the flowchart of this approach. Typically, the algorithm consists of three basic steps:

1. **Background maintenance:** the initial background model is obtained even if there are moving foreground objects in the field of view, such as walking people, moving cars, etc. In the algorithm, the frequency ratios of the intensity values for each pixel at the same position in the frames are calculated using N frames to distinguish moving pixels from stationary ones, and the intensity values for each pixel with the biggest ratios are incorporated to model the background scene. We extend this idea to the RGB color space. Each component of RGB is respectively treated with the same scheme. A RGB histogram is created to record the frequency of each component of the pixel. This process is illustrated in Figure 5.15.

   The RGB component value with the biggest frequency ratio estimated in the histogram is assigned as the corresponding component value of the pixel in the background model. In order to maintain the Background model we reinitialize

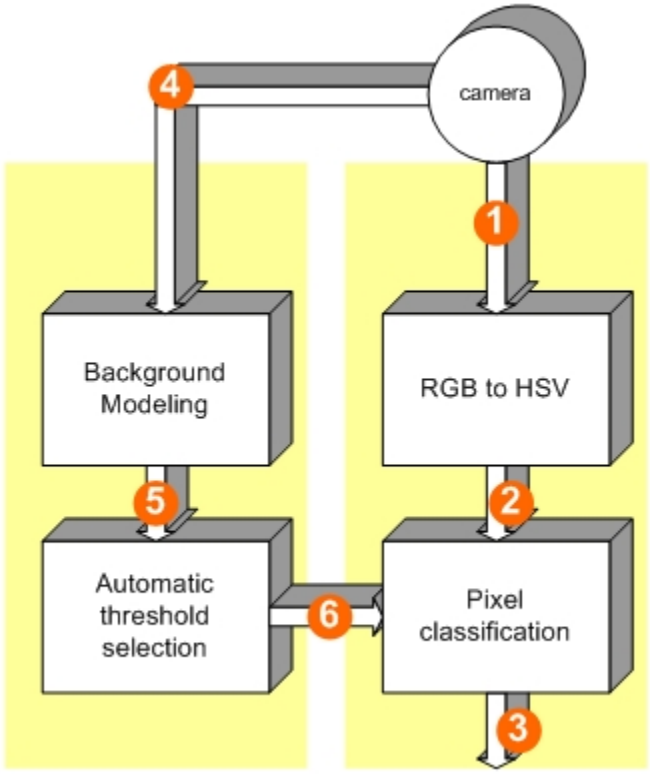


Figure 5.14: Flowchart of the Deterministic non-model based approach. Periodically, each N frames are used to compute, for each pixel, the most representative RGB pixel values with higher frequencies f^R, f^G and f^B (5). For each frame (1), a conversion from RGB to HSV is applied (2), then each pixel is classified using two stages pixel process into three classes $C(i) \in \{F, B, S\}$ (3) using a decision rule based on thresholds τ_H, τ_s, R, M, M_{min} and $Mn(x, y)$ periodically computed (6).

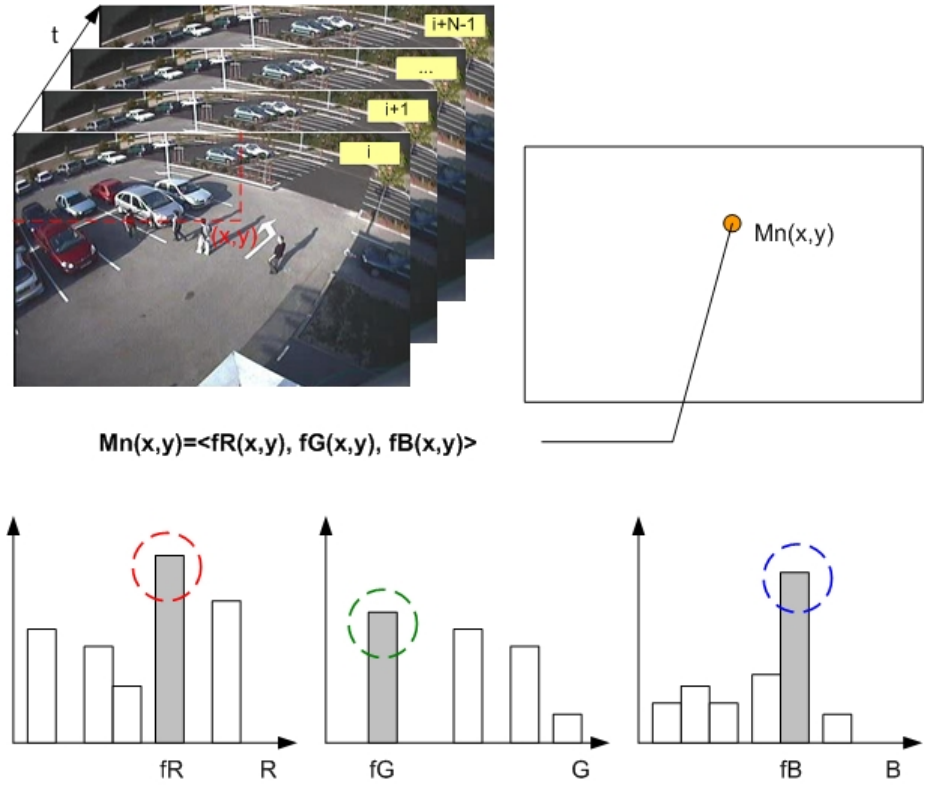


Figure 5.15: RGB histogram of pixel (x, y) over the learning period for background initiation.

it periodically each N frames (N = 100-200 frames) to adapt to the illumination changes and scene geometry changes in the background scene.

Figure 5.16: (A) frame 106, (B) frame 202, (C) estimated background

2. **Pixel classification:** pixel classification is a two-stage process. The first stage aims at segmenting the foreground pixels. The distortion of brightness for each pixel between the incoming frame and the reference image is computed. This

stage is performed according to the following equation:

$$F(x,y) = \begin{cases} 1 & if \quad |I^v(x,y) - B^v(x,y)| \geq 1.1 M_n(x,y) \\ & \quad \vee |I^v(x,y) - B^v(x,y)| > M_{min} \\ 0 & otherwise \end{cases} \tag{5.37}$$

The second stage consists in the segmentation of the shadow pixels. This is done by introducing the hue and saturation information of the candidate pixels, as well as the division of brightness between the pixels and the same pixels in background image. Thus, the shadow points are classified by the following decision procedure:

$$S(x,y) = \begin{cases} 1 & if \quad I^v(x,y) - B^v(x,y) < M \\ & \quad \wedge L < |I^v(x,y)/B^v(x,y)| < R \\ & \quad \wedge |I^H(x,y) - B^H(x,y)| > \tau_H \\ & \quad \wedge I^s(x,y) - B^s(x,y) < \tau_s \\ 0 & otherwise \end{cases} \tag{5.38}$$

3. **Threshold selection:** equation 5.37 introduces two thresholds: $M_n(x,y)$ and M_{min}. $M_n(x,y)$ is the mean value of the distortion of brightness for the pixel at the position (x,y) over the last N frames. M_{min} is a minimum mean value which is introduced as a noise threshold to prevent the mean value from decreasing below a minimum should the background measurements remain strictly constant over a long period of time. M_{min} is set at 40.

   Equation 5.38 involves four thresholds: M, R, τ_H and τ_s. R is the ratio between pixels when illuminated and the same pixels. This ratio is linear according to [RE95]. We experimentally fixed the value of R to 2.2 which is coherent with the range that has been experimentally found by Chen et al. [CL04]. L and M have been empirically set respectively to 1.2 and 20.

   τ_H and τ_s are selected threshold values used to measure the similarities of the hue and saturation between the background image and the current observed image. τ_H is set at 30 and τ_s is set at -10.

5.5 High level feedback to improve detection methods

Background models are usually designed to be task independent, and this often means that they can use very little high-level information. Several approaches to incorporating information about foreground objects into background maintenance have been proposed. They may be broadly separated into two categories: probabilistic frameworks that jointly model scene and foreground object evolution, and systems consisting of separate modules for scene modeling and high level inference.

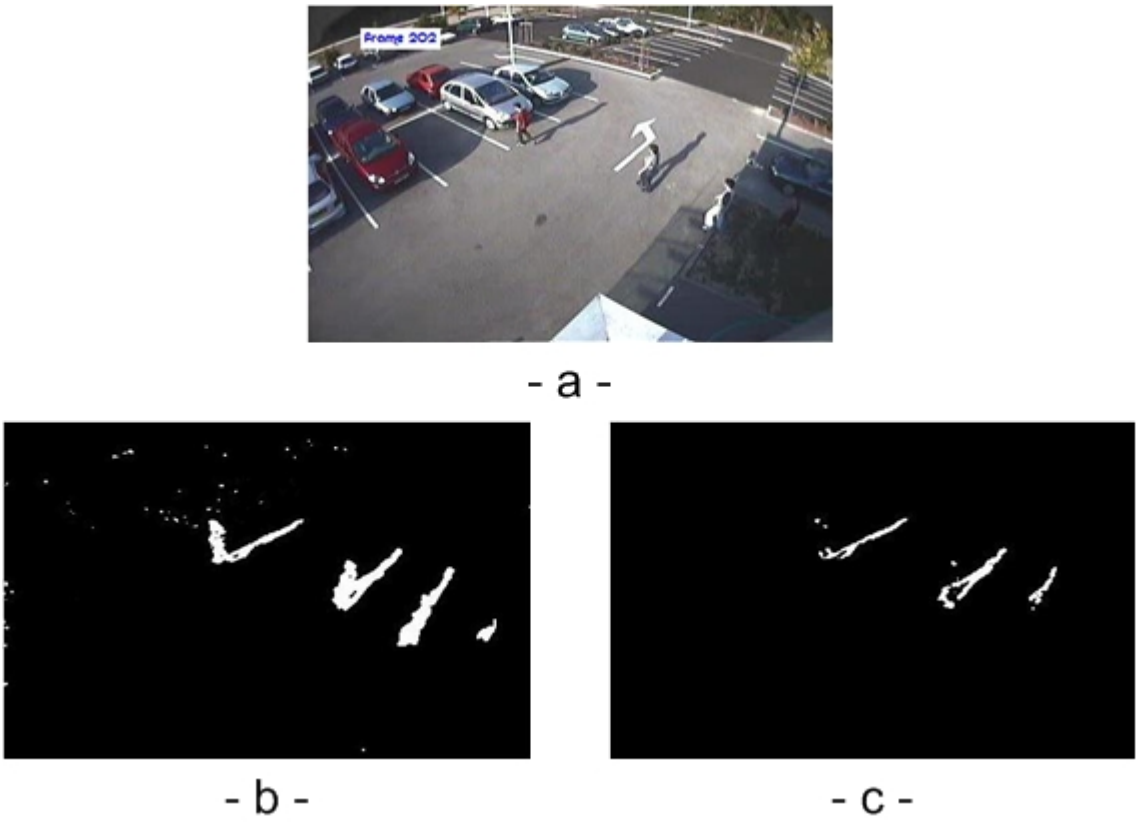


Figure 5.17: Result of deterministic non-model based approach:(a) current frame, (b) without shadow detection result (c) shadows

5.5.1 The modular approach

The usage of high level information in the background modeling framework is present at the model update level. Instead of updating the background model using the classification result, the background model is updated using the classification result postprocessed using the high level algorithms (tracking, shadow detection...).

The advantage of the modular approach is that it conserves the known algorithms for adaptive background differencing. It only intercepts the model update step. It provides a corrected mask using the high level information. Thus, the usage of the previous algorithms is ease to take in hand. Otherwise, the usage of the high level information depends on the sensibility of the subsequent background model.

In our work we use the tracking and shadow detection as high level information provider. We do not explain all the implementation in the present thesis (this part is well explained in the library developed in C++ for implementing the different algorithms), we concentrate on the comparison of several algorithms and the integration to the rest of the framework.

5.6 Performance evaluation

Performance evaluation is an important issue in the development of object detection algorithms. Quantitative measures are useful for optimizing, evaluating and comparing different algorithms. For object detection, we believe there is no single measure for quantifying the different aspects of performance. In this section, we present a set of performance measures for determining how well a detection algorithm output matches the ground truth.

5.6.1 Ground truth generation

A number of semi-automatic tools are currently available for generating ground truth from pre-recorded video. The Open development environment for evaluation of video systems (ODViS) [JWSX02] allows the user to generate ground truth, incorporate new tracking algorithm into the environment and define any error metric through a graphical user interface. The Video Performance Evaluation Resource (ViPER) [DM00] is directed more toward performance evaluation for video analysis systems. It also provides an interface to generate ground truth, metrics for evaluation and visualization of video analysis results.

In our experiments the comparison is based on manually annotated ground truth. We used to draw a bounding box around each individual and also around each group of individuals. Individuals are labeled only once they start moving. Groups are defined as two or more individuals that interact. Groups cannot be determined only by analyzing the spatial relations between individuals which makes the detection and tracking very difficult for artificial systems. For this reason, we decide to restrict the evaluation only to individual bounding boxes. On the other hand we labeled the shape of the given individual. We define two labels: one label for the individual and another for the moving cast shadow. This two-level labeling (pixel based and object based) is useful to compare the object detection algorithms using adaptive background differencing.

5.6.2 Datasets

We evaluated the performance of the different algorithms on a collection of datasets for outdoor and indoor scenes. The outdoor scenes are mainly focused on people moving in parking. The acquisition took place in INRIA (Grenoble). Figure 5.18 shows an example of the parking scene with the appropriate visual labeling (we can distinguish the main object and the casted shadow).

Figure 5.19 shows the well known example of indoor dataset: the intelligent room. the usage of this dataset was useful to compare our results with published results and to verify the correctness of other results as well.

Figure 5.18: Example frame of the parking ground truth

Figure 5.19: Example frame of intelligent room ground truth

5.6.3 Evaluation metrics

In order to evaluate the performance of object detection algorithms, we adopted some of the performance metrics proposed in [HA05] and [BER03]. We used also other metrics introduced in [VJJR02]. These metrics and their significance are described below.

Let $G^{(t)}$ be the set of ground truth objects in a single frame t and let $D^{(t)}$ be the set of the output boxes produced by the algorithm. $N_{G^{(t)}}$ and $N_{D^{(t)}}$ are their respective values in frame t.

1. **Precision(FAR) and Recall(TRDR):**

$$TRDR = \frac{TP}{TP + FN} \tag{5.39}$$

$$FAR = \frac{TP}{TP + FP} \tag{5.40}$$

Where a True Positive (TP) is defined as a ground truth point that is located within the bounding box of an object detected by the algorithm. A False negative (FN) is a ground truth point that is not located within the bounding box of any object detected by the algorithm. A False positive (FP) is an object that is detected by the algorithm that has not a matching ground truth point. A correct match is registered, when the bounding boxes of the targets A_{obs} and A_{truth} overlap T.

$$\frac{A_{obs} \cap A_{truth}}{A_{obs} \cup A_{truth}} \geq T \tag{5.41}$$

2. **Area-Based Recall for Frame(ABRF):**
This metric measures how well the algorithm covers the pixel regions of the ground truth. Initially it is computed for each frame and it is the weighted average for the whole data set. Let $U_{G^{(t)}}$ and $U_{D^{(t)}}$ be the spatial union of the

Method	Room	Parking
Recall at T=50%		
LOTS	0.3800	0.7500
W4	0.3900	0.0789
MGM	0.3799	0.5132
SGM	0.0000	0.0000
BBS	0.0000	0.0000
BBS-R	0.0092	0.0078
Precision at T=50%		
LOTS	0.0800	0.0600
W4	0.0115	0.0001
MGM	0.0000	0.0088
SGM	0.0000	0.0000
BBS	0.0000	0.0000
BBS-R	0.0092	0.0022

Table 5.2: Comparison of recall and precision of the algorithms evaluated on the indoor and outdoor datasets

boxes in $G^{(t)}$ and $D^{(t)}$:

$$U_{G^{(t)}} = \bigcup_{i=1}^{N_{G^{(t)}}} G_i^{(t)} \tag{5.42}$$

$$U_{D^{(t)}} = \bigcup_{i=1}^{N_{D^{(t)}}} D_i^{(t)} \tag{5.43}$$

For a single frame t, we define $Rec(t)$ as the ratio of the detected areas in the ground truth with the total ground truth area:

$$Rec(t) = \begin{cases} undefined & \text{if } U_{G^{(t)}} = \emptyset \\ \frac{|U_{G^{(t)}} \cap U_{D^{(t)}}|}{|U_{G^{(t)}}|} & otherwise \end{cases} \tag{5.44}$$

$$OverallRec = \begin{cases} undefined & \text{if } \sum_{t=1}^{N_f} |U_{G^{(t)}}| = 0 \\ \frac{\sum_{t=1}^{N_f} |U_{G^{(t)}}| \times Rec(t)}{\sum_{t=1}^{N_f} |U_{G^{(t)}}|} & otherwise \end{cases} \tag{5.45}$$

where N_f is the number of frames in the ground-truth data set and the $|$ operator denotes the number of pixels in the area.

3. **Area-Based Precision for Frame(ABPF):**
This metric measures how well the algorithm minimized false alarms. Initially

Method	Room	Parking
LOTS	0.3731	0.7374
W4	0.8464	0.6614
MGM	0.5639	0.8357
SGM	0.6434	0.5579
BBS	0.8353	0.9651
BBS-R	0.7000	0.5911

Table 5.3: ABRF metric evaluation of the different algorithms

it is computed for each frame and it is the weighted average for the whole data set. Let $U_{G^{(t)}}$ and $U_{D^{(t)}}$ be the spatial union of the boxes in $G^{(t)}$ and $D^{(t)}$:

$$U_{G^{(t)}} = \bigcup_{i=1}^{N_{G^{(t)}}} G_i^{(t)} \tag{5.46}$$

$$U_{D^{(t)}} = \bigcup_{i=1}^{N_{D^{(t)}}} D_i^{(t)} \tag{5.47}$$

For a single frame t, we define $Prec(t)$ as the ratio of the detected areas in the ground truth with the total ground truth area:

$$Prec(t) = \begin{cases} undefined & \text{if } U_{D^{(t)}} = \emptyset \\ 1 - \frac{|U_{G^{(t)}} \cap U_{D^{(t)}}|}{|U_{D^{(t)}}|} & otherwise \end{cases} \tag{5.48}$$

OverallPrec is the weighted average precision of all the frames.

$$OverallPrec = \begin{cases} undefined & \text{if } \sum_{t=1}^{N_f} |U_{D^{(t)}}| = 0 \\ \frac{\sum_{t=1}^{N_f} |U_{G^{(t)}}| \times Rec(t)}{\sum_{t=1}^{N_f} |U_{D^{(t)}}|} & otherwise \end{cases} \tag{5.49}$$

where N_f is the number of frames in the ground-truth data set and the $|$ operator denotes the number of pixels in the area.

Method	Room	Parking
LOTS	0.1167	0.3883
W4	0.5991	0.7741
MGM	0.5652	0.6536
SGM	0.8801	0.8601
BBS	0.9182	0.9526
BBS-R	0.5595	0.4595

Table 5.4: ABPF metric evaluation of the different algorithms

4. **Average Fragmentation(AF):**
This metric is intended to penalize an algorithm for multiple output boxes covering a single ground-truth object. Multiple detections include overlapping and non-overlapping boxes. For a ground-truth object $G_i^{(t)}$ in frame t, the fragmentation of the output boxes overlapping the object $G_i^{(t)}$ is measured by:

$$Frag(G_i^{(t)}) = \begin{cases} undefined & \text{if } N_{D^{(t)} \cap G_i^{(t)}} = 0 \\ \frac{1}{1+log_{10}(N_{D^{(t)} \cap G_i^{(t)}})} & \text{otherwise} \end{cases} \tag{5.50}$$

where $N_{D^{(t)} \cap G_i^{(t)}}$ is the number of output boxes in $D^{(t)}$ that overlap with the ground truth object $G_i^{(t)}$.

Overall fragmentation is defined as average fragmentation for all ground-truth objects in the entire data set.

5. **Average Object Area Recall(AOAR):**
This metric is intended to measure the average area recall of all the ground-truth objects in the data set. The recall for an object is the proportion of its area that is covered by the algorithm's output boxes. The objects are treated equally regardless of size.

For a single frame t, we define $Recall(t)$ as the average recall for all the objects in the ground truth $G^{(t)}$:

$$Recall(t) = \frac{\sum_{\forall G_i^{(t)}} ObjectRecall(G_i^{(t)})}{N_{G^{(t)}}} \tag{5.51}$$

Where

$$ObjectRecall(G_i(t)) = \frac{|G_i^{(t)} \cap U_{D^{(t)}}|}{|G_i^{(t)}|} \tag{5.52}$$

$$OverallRecall(G_i(t)) = \frac{\sum_{t=1}^{N_f} N_{G^{(t)}} \times Recall(t)}{\sum_{t=1}^{N_f} |N_{G^{(t)}}|} \tag{5.53}$$

Method	Room	Parking
LOTS	0.1500	0.7186
W4	0.5555	0.6677
MGM	0.2920	0.8468
SGM	0.4595	0.5418
BBS	0.7026	0.9575
BBS-R	0.3130	0.1496

Table 5.5: AOAR metric evaluation of the different algorithms

6. **Average Detected Box Area Precision(ADBAP):**
This metric is a counterpart of the previous metric 5.51 where the output boxes are examined instead of the ground-truth objects. Precision is computed for each output box and averaged for the whole frame. The precision of a box is the proportion of its area that covers the ground truth objects.

$$Precision(t) = \frac{\sum_{\forall D_i^{(t)}} BoxPrecision(D_i^{(t)})}{N_{D^{(t)}}} \tag{5.54}$$

$$BoxPrecision(D_i^{(t)}) = \frac{|D_i^{(t)} \cap U_{G^{(t)}}|}{|D_i^{(t)}|} \tag{5.55}$$

$$OverallPrecision = \frac{\sum_{t=1}^{N_f} N_{D^{(t)}} \times Precision(t)}{\sum_{t=1}^{N_f} |N_{D^{(t)}}|} \tag{5.56}$$

Method	Room	Parking
LOTS	0.0053	0.1453
W4	0.0598	0.0417
MGM	0.0534	0.0177
SGM	0.0133	0.0192
BBS	0.0048	0.0002
BBS-R	0.1975	0.0192

Table 5.6: ADBAP metric evaluation of the different algorithms

7. **Localized Object Count Recall(LOCR):**
In this metric, a ground-truth object is considered as detected if a minimum proportion of its area is covered by the output boxes. Recall is computed as the ratio of the number of detected objects with the total number of ground-truth objects.

$$LocObjRecall(t) = \sum_{\forall G_i^{(t)}} ObjDetect(G_i^{(t)}) \tag{5.57}$$

$$ObjDetect(G_i^{(t)}) = \begin{cases} 1 & \text{if } \frac{|G_i^{(t)} \cap U_{D^{(t)}}|}{|G_i^{(t)}|} > OverlapMin \\ 0 & \text{otherwise} \end{cases} \tag{5.58}$$

$$OverallLocObjRecall = \frac{\sum_{t=1}^{N_f} LocObjRecall(t)}{\sum_{t=1}^{N_f} N_{G_i^{(t)}}} \tag{5.59}$$

Method	Room	Parking
LOTS	0.0056	0.0190
W4	0.0083	0.0127
MGM	0.0083	0.0253
SGM	0.0056	0.0190
BBS	0.0083	0.0253
BBS-R	0.0028	0.0063

Table 5.7: LOCR metric evaluation of the different algorithms

8. **Localized Output Box Count Precision(LOBCP):**
This is a counterpart of metric 5.56. It counts the number of output boxes that significantly covered the ground truth. An output box $D_i^{(t)}$ significantly covers the ground truth if a minimum proportion of its area overlaps with $U_{G^{(t)}}$.

$$LocBoxCount(t) = \sum_{\forall D_i^{(t)}} BoxPrec(D_i^{(t)}) \tag{5.60}$$

$$BoxPrec(D_i^{(t)}) = \begin{cases} 1 & \text{if } \frac{|D_i^{(t)} \cap U_{G^{(t)}}|}{|D_i^{(t)}|} > OverlapMin \\ 0 & \text{otherwise} \end{cases} \tag{5.61}$$

$$OverallOutputBoxPrec = \frac{\sum_{t=1}^{N_f} LocBoxCount(t)}{\sum_{t=1}^{N_f} N_{D_i^{(t)}}} \tag{5.62}$$

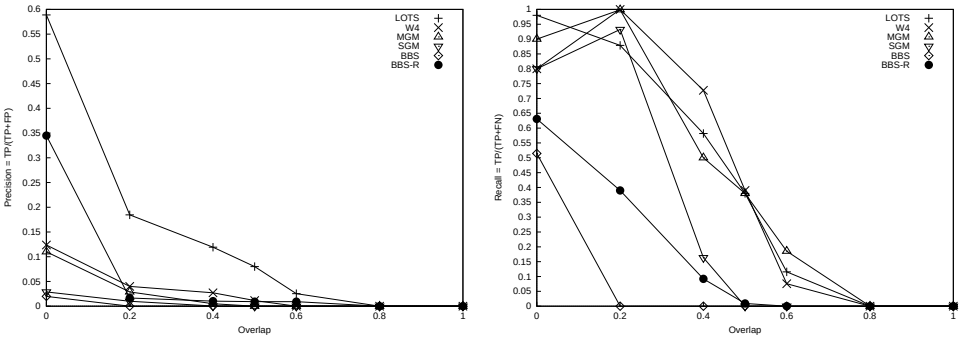
Method	Room	Parking
LOTS	0.5919	0.1729
W4	0.0592	0.0427
MGM	0.0541	0.0206
SGM	0.0124	0.0186
BBS	0.0043	0.0001
BBS-R	0.1900	0.0174

Table 5.8: LOCBP metric evaluation of the different algorithms

5.6.4 Experimental results

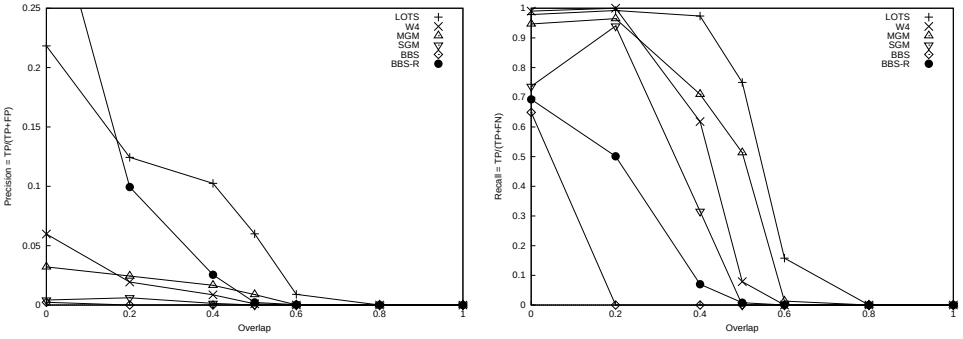
Figure 5.20(a) and figure 5.20(b) display respectively precision and recall of different methods evaluated on the experimental indoor dataset. Figure 5.21(a) and figure 5.21(b) display respectively precision and recall of different methods evaluated on the experimental outdoor dataset. Table 5.2 shows precision and recall with an overlap requirement of T=50% (see eq 5.41).

According to those results we can introduce the following remarks: the simple method BBS gives the lower benchmark on precision, since it produces a very high



(a) precision with respect to overlap (b) recall with respect to overlap

Figure 5.20: Room sequence



(a) precision with respect to overlap (b) recall with respect to overlap

Figure 5.21: Parking sequence

number of false detections. The method MGM has the best recall for high overlap thresholds ($T > 0.6$) and for low overlap thresholds ($T < 0.2$) LOTS get the best one for indoor dataset, but for outdoor dataset (parking) LOTS has already the best recall. Which means that the size estimates are quite good. This is due to the connected component analysis of the methods.

In the case of indoor scene, BBS-R performs better than BBS, and BBS-R has also higher precision for $T < 0.4$ but it still has lower recall. This method seems as well as BBS not appropriate for the task in the indoor scenes. On the other side it performs better in the case of outdoor scenes.

The new method performs better than the original BBS algorithm. It outperforms SGM and W4, but it still not two performant as the complex algorithms MGM and LOTS. Thus, the new approach is modular, we can imagine to apply the same method to those complex algorithms.

For some of the evaluated sequences there was the problem of sudden camera jitter, which can be caused by a flying object or strong wind. This sudden motion causes the motion detector to fail and identify the whole scene as a moving object, although the system readjusts itself after a while but this causes a large drop in the performance.

5.6.5 Comments on the results

The evaluation results shown above were affected by some very important factors:

1. For some of the evaluated sequences there was the problem of sudden camera jitter, which can be caused by a flying object or strong wind. This sudden motion cause the motion detector to fail and identify the whole scene as a moving object, although the system readjusts itself after a while but this cause a large drop in the performance.

2. The number of objects in the sequence alters the performance significantly where the optimum performance occurs in the case of a few independent objects in the scene. Increasing the number of objects increases the dynamic occlusion and loss of track.

Chapter 6

Machine learning for visual object-detection

Contents

6.1 Introduction

In this chapter we present a machine learning framework for visual object-detection. We called this framework LibAdaBoost as an abreviation of *Library for Adaptive Boosting*. This framework is specialized in boosting.

LibAdaBoost supports visual data types, and provides a set of tools which are ready for use by computer vision community. These tools are useful for the generation of visual detectors using state of the art boosting algorithms.

This chapter is organized as follows: section 6.2 gives a general introduction for boosting. Section 6.3 describes the usage of boosting for visual object-detection. Section 6.4 introduces LibAdaBoost and its software architecture and the subsequent workflows for learning tasks, validation tasks and test tasks. Section 6.5 describes use cases for examples of visual object-detection with LibAdaBoost. Each use case is described by its learning part, its validation part and its test on benchmarked datasets. Section 6.6 concludes the chapter.

6.2 The theory of boosting

Boosting has been presented as a very effective technique to help learning algorithms. This method allows, as its name implies, to boost the hypotheses given by a weak learner. AdaBoost, one of the most interesting boosting algorithm, has first been proposed by Schapire and Freund [FS95a] in the 90's. It has further grown in the works of Schapire and Singer [SS98] where it was generalized making the first version a specific one of the second. A good introduction to AdaBoost is also available in [FS99].

Boosting has been used first for visual object detection by Viola and Jones [VJ01b], for detecting faces since 2000. Other works for detection of pedestrians [VJS03], cars [KNAL05], license plates [ZJHW06], and others [ASG05]. This increasing usage of Boosting motivated us for developing a unified platform for different boosting algorithms, data types, and subsequent weak learners and classifiers.

6.2.1 Conventions and definitions

In this section we will introduce some definitions related to learning process. These definitions are useful for easily understanding section 6.2.2. Whether we are talking about machines or humans the learning process could be decomposed into two main parts as shown in Figure 6.1. First of all one has to be able to retain previously seen

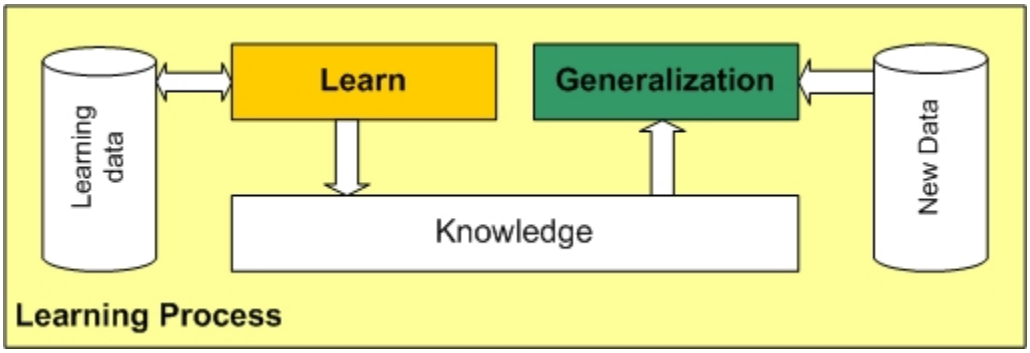


Figure 6.1: Learning process: first step consists in retaining previously seen data so that one is able to identify it when it is seen again. The second step consists in applying the knowledge, gained from previous step, to new data, this step is called generalization.

data so that it is able to identify it when it is seen again. The second step consists in applying the knowledge, gained from previous step, to new data. This fact is known as generalization and is the main goal of the learner. Thus we introduce the definition of the generalization error in Definition 1.

Definition 1 (Generalization error)**.** *The generalization error is the error rate of an algorithm trying to apply its expertise to new samples.*

Formally we can define a learning algorithm as in Definition 2.

Definition 2 (Learning algorithm)**.** *A learning algorithm is a procedure which can reduce the error on a set of training data [DHS00] as well as have the capacity of minimizing the generalization error.*

The training set is the input data for the learning algorithm. It is defined as in Definition 3.

Definition 3 (Training set)**.** *A training set S is defined by $(x_1, y_1), ..., (x_n, y_n)$ where each x_i belongs to some* ***domain*** *or* ***instance space*** *X, and each label y_i belongs to some* ***label set*** *Y.*

In this work we focus on learning using classifiers defined in Definition 4. An example of a classifier would be a voting system which, given an email, would be able to classify it as spam email or safe email.

Definition 4 (Knowledge)**.** *We define the knowledge as a classifier $C : X \to Y$ which calculates, for a given sample x, the appropriate label y.*

In the following sections we will be using binary classifiers. A binary classifier is defined in Definition 5.

Definition 5 (Binary Classifier)**.** *C is a binary classifier if $Y = \{-1, +1\}$.*

6.2.2 Boosting algorithms

To understand *boosting* we will use the horse-racing gambler example which is taken from [FS99].

A horse-racing gambler, hoping to maximize his winnings, decides to create a computer program that will accurately predict the winner of a horse race based on the usual information (number of races recently won by each horse, betting odds for each horse, etc.). To create such a program, he asks a highly successful expert gambler to explain his betting strategy. Not surprisingly, the expert is unable to articulate a grand set of rules for selecting a horse. On the other hand, when presented with the data for a specific set of races, the expert has no trouble coming up with a "rule of thumb" for that set of races (such as, "Bet on the horse that has recently won the most races" or "Bet on the horse with the most favored odds"). Although such a rule of thumb, by itself, is obviously very rough and inaccurate, it is not unreasonable to expect it to provide predictions that are at least a little bit better than random guessing. Furthermore, by repeatedly asking the expert's opinion on different collections of races, the gambler is able to extract many rules of thumb.

In order to use these rules of thumb to maximum advantage, there are two problems faced by the gambler: first, how should he choose the collections of races presented to the expert so as to extract rules of thumb from the expert that will be the most useful? Second, once he has collected many rules of thumb, how can they be combined into a single, highly accurate prediction rule?

Boosting refers to a general and provably effective method of producing a very accurate prediction rule by combining rough and moderately inaccurate rules of thumb in a manner similar to that suggested above.

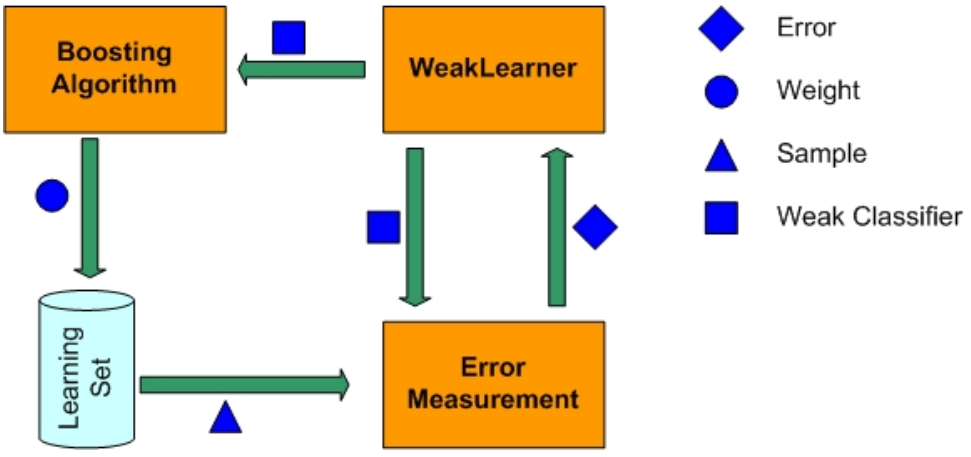


Figure 6.2: Boosting algorithm framework

Historical overview

The following summary is based on [FS99]: originally, boosting started from a theory of machine learning called "Probably Approximately Correct" (PAC) [Val84, KV94]. This model allows us to derive bounds on estimation error for a choice of model space and given training data. In [KV88, KV89] the question was asked if a "weak" learning algorithm, which works just slightly better than random guessing (according to the PAC model) can be made stronger (or "boosted") to create a "strong" learning algorithm. The first researcher to develop an algorithm and to prove its correctness is Schapire [Sch89]. Later, Freund [Fre90] developed a more efficient boosting algorithm that was optimal but had several practical problems. Some experiments with these algorithms were done by Drucker, Schapire and Simard [DSS93] on an OCR task.

6.2.3 AdaBoost

The AdaBoost algorithm was introduced in 1995 by Freund and Schapire [FS95a]. AdaBoost solved many of the practical problems that existed in the previous boosting algorithms [FS95a]. In Algorithm 1 we bring the original AdaBoost algorithm. The algorithm takes as input a training set $\{(x_1, y_1), \ldots (x_m, y_m)\}$ where each x_i belongs to some *domain* X and each label y_i is in some label set Y. To our purposes we need only binary classification and thus can assume $Y = \{+1, -1\}$. Generalization to the multi-class case is out of the focus of this thesis. AdaBoost calls a given *weak* learning

Algorithm 1 The Adaboost algorithm in its original form

Require: a sequence of N labeled samples $\{(x_1, y_1), \ldots, (x_m, y_m)\}$ where $x_i \in X$ and $y_i \in Y = \{-1, +1\}$

1: Initialize the weight vector: $D_1(i) = \frac{1}{m}$
2: **for** $t = 1$ to T **do**
3: Train weak learner, providing it with the distribution D_t.
4: Get weak classifier (weak hypothesis) $h_t : X \to \{-1, +1\}$ with error
$$\epsilon_t = Pr_{i \sim D_t}[h_t(x_i) \neq y_i].$$
5: Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$
6: Update:
$$D_{t+1}(i) = D_t(i) \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases}$$
7: Normalize the weights so that D_{t+1} will be a distribution:
$$\sum_{i=1}^{N} D_{t+1}(i) = 1.$$
8: **end for**

Ensure: Strong classifier $H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$

algorithm repeatedly in a series of "cycles" $t = 1 \ldots T$. The most important idea of the algorithm is that it holds at all times a distribution over the training set (i.e. a weight for each sample). We will denote the weight of this distribution on training sample (x_i, y_i) on round t by $D_t(i)$. At the beginning, all weights are set equally, but

on each round, the weights of incorrectly classified samples are increased so that the weak learner is forced to focus on the hard samples in the training set.

The role of the weak learner is to find a weak rule (classifier) $h_t : X \rightarrow Y$ appropriate for the distribution D_t. But what is "appropriate"? The "quality" of a weak classifier is measured by its error, according to the weights:

$$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i) \tag{6.1}$$

The error is measured, of course, with respect to the distribution D_t on which the weak learner was trained, so each time the weak learner finds a different weak classifier. In practice, the weak learner is an algorithm - doesn't matter which algorithm - that is allowed to use the weights D_t on the training samples.

Coming back to the horse-racing example given above, the instances x_i correspond to descriptions of horse races (such as which horses are running, what are the odds, the track records of each horse, etc.) and the label y_i gives a binary outcome (i.e., if your horse won or not) of each race. The weak classifiers are the rules of thumb provided by the expert gambler, where he chooses each time a different set of races, according to the distribution D_t. Algorithm 2 shows the AdaBoost algorithm running in the head of the gambler.

Algorithm 2 The Adaboost algorithm running in the head of the gambler

1: If I look at the entire set of races I've seen in my life, I can tell that it is very important that the other horses (other than the one I gamble on) will not be too strong. This is an important rule.
2: I will now concentrate on the races where my horse was the strongest horse, but nevertheless it didn't win. I recall that in some of these races, it was because there was a lot of wind, and even a strong horse might be sensitive to wind. This cannot be known in advance about a certain horse.
3: In another case the horse I chose was not the strongest, but even though, it won. This was because it was in the outer lane, and it is easier to run there.

Ensure: Resulting strong classifier
The other horses (other than the one I gamble on) should not be too strong. Importance of rule: 7 (on a scale of 1 to 10).
These should not be wind in the day of the race. Importance: 3.
The horse should be in the outer lane. Importance: 2.

Note that the importances of the weak hypothesis are calculated according to their error in the execution of AdaBoost. This corresponds to the α of the rule in the algorithm (see Algorithm 1). Once the weak hypothesis h_t has been received from the weak learner, AdaBoost chooses a parameter α_t. Intuitively, α_t measures the importance that is assigned to h_t.

Note that $\alpha_t \geq 0$ if $\epsilon_t < \frac{1}{2}$ (which we can assume without loss of generality), and that α_t gets larger as ϵ_t gets smaller. The distribution D_t is then updated using

the rule shown in Algorithm 1. This rule actually increases the weight of samples misclassified by h_t , and decreases the weight of correctly classified samples. Thus, the weight tends to concentrate on "hard" samples.

The *final hypothesis* H is a weighted majority vote of the T weak hypotheses where α_t is the weight assigned to h_t .

Variations on AdaBoost

AdaBoost as it was showed in the previous section could only classify binary discrete problems. Various extensions have been created (for multi-class and multi-label) such as the ones described in [FS95b] known as AdaBoost.M1, AdaBoost.M2. Simpler forms of these can also be found in [SS98]. There have also been several very important additions to AdaBoost. one of the most interesting used in language recognition problems is prior knowledge [RSR+02]. It enables to add human knowledge to maximize the classification properties of the algorithm. It basically gives some knowledge to the learner before training. This is showed to be especially useful when the number of training samples is limited.

6.2.4 Weak classifier

A weak classifier depends on the Domain X. In this thesis we focus on the Visual Domain. Therefore, Definition 6 represents the description of the weak classifier in a generic boosting framework. In section 6.3.3 we present a set of weak classifiers for the visual domain.

Definition 6 (Weak classifier)**.** *Let X as given Domain. Let Y as a set of labels. We consider a binary classification case :$Y = \{-1, +1\}$. A weak classifier $h : X \to Y$ is a classifier that is used as an hypothesis in the boosting framework.*

6.2.5 Weak learner

The AdaBoost algorithm needs a weak classifier provider, this is called the weak learner. As described in Definition 8 the weak learner is a learning algorithm which will provide a "good" feature at each boosting cycle, when the goodness is measured according to the current weights of the samples. Obviously, choosing the best weak classifier at each boosting cycle cannot be done by testing all the possibilities in the weak classifier's space. Therefore, a partial search (selective search) strategy can be used to search a good weak classifier. This weak classifier is not always the best according to the whole space. Thus, partial strategies can be based on genetic-like approaches or other heuristics.

Definition 7 (Weak Learner)**.** *A Weak Learner is a learning algorithm responsible of the generation of a weak classifier (weak hypothesis). A weak learner is independent of the Domain X. AdaBoost needs a weak learner with an error rate better than random.*

In this section we will describe an improvement of a Genetic-Like algorithm that was first introduced by Abramson [Abr06]. In our work we implemented a generic implementation of the algorithm that is completely independent of the domain X.

Genetic-like weak learner

The genetic-like algorithm, given in Algorithm 3, maintains a set of weak classifiers which are initialized as random ones. At each step of the algorithm, a new "generation" of weak classifiers is produced by applying a set of mutations on each of the weak classifiers. All the mutations are tested and the one with the lowest error might replace the "parent" if it has a lower error. In addition, some random weak classifiers are added at each step.

Algorithm 3 The Genetic-Like algorithm from [Abr06]

Require: An error measurement Γ that matches an error to every weak classifier, a number G of generations to run, and a number N of "survivors" at each generation.

1: Let Υ be a generator of random weak classifiers. Initialize the first generation's weak classifiers vector c^1 as:
$c_i^1 \leftarrow \Upsilon \qquad i = 1 \dots N$
2: **for** $g = 1$ to G **do**
3: Build the next generation vector c^{g+1} as:
$c_i^{g+1} \leftarrow c_i^g \qquad i = 1 \dots N$
4: **for** m as Mutation in MutationSet **do**
5: **if** $\Im^m(c_i^g)$ is valid and $\Gamma(\Im^m(c_i^g)) < \Gamma(c_i^{g+1})$ **then**
6: $c_i^{g+1} \leftarrow \Im^m(c_i^g) \qquad i = 1 \dots N$
7: **end if**
8: **end for**
9: Enlarge the vector C^{g+1} with N additional weak classifiers, chosen randomly:
$c_i^{g+1} \leftarrow \Upsilon \qquad i = N+1, \dots, 2N$
10: Sort the vector C^{g+1} such that
$\Gamma(c_1^{g+1}) \leq \Gamma(c_2^{g+1}) \leq \dots \leq \Gamma(c_{2N}^{g+1})$
11: **end for**
12: **return** the weak classifier c_1^{G+1}.

The genetic algorithm continues to evolve the generations until there is no improvement during G consecutive generations. An improvement for that matter is of course a reduction in the error of the best feature in the generation.

Definition 8 (Error measurment)**.** *Given a learning sample set Ω, and a classifier Ψ. An error measurement relative to Ω is a function Γ_Ω that matches for Ψ a weighted error relative to the weights in Ω.*

In our work, we implemented an advanced instance of the previous weak learner (Algorithm 3). The new advanced genetic-like algorithm (AGA) is shown in Algo-

Algorithm 4 AGA: Advanced version of Genetic-like Algorithm

Require: A learning sample set Ω and its associated error measurement Γ_Ω (see Definition 8).

Let C as a vector of N classifiers. C_k is the k^{th} element in C.
Let C^g the status of C at the g^{th} iteration.
Let Υ be a generator of random weak classifiers.
$C_i^1 \leftarrow \Upsilon$, $i = 1 \dots N$
$I_{NoImprovement} \leftarrow 0, g \leftarrow 1$
$\Psi \leftarrow C_1^1$
while true **do**
  $g \leftarrow g + 1$
  Sort the vector C^g such that $\Gamma_\Omega(C_1^g) \leq \Gamma_\Omega(C_2^g) \leq \cdots \leq \Gamma_\Omega(C_N^g)$
  if $\Gamma_\Omega(\Psi) \geq \Gamma_\Omega(C_1^g)$ **then**
    $I_{NoImprovement} \leftarrow 0$
    $\Psi \leftarrow C_1^g$
  else
    $I_{NoImprovement} \leftarrow I_{NoImprovement} + 1$
  end if
  if $I_{NoImprovement} = N_{maxgen}$ **then**
    return Ψ
  end if
  for $i = 1$ to N_{best} **do**
    if $\Gamma(\Im^m(C_i^g)) < \Gamma(C_i^g)$ **then**
      $C_i^g \leftarrow \Im^m(C_i^g)$
    end if
  end for
  for $i = N_{best} + 1$ to $N_{best} + N_{looser}$ **do**
    if $\Gamma(\Upsilon(C_i^g)) < \Gamma(C_i^g)$ **then**
      $C_i^g \leftarrow \Upsilon(C_i^g)$
    end if
  end for
  for $i = N_{best} + N_{looser} + 1$ to N **do**
    $C_i^g \leftarrow \Upsilon$
  end for
end while
return Ψ

Ensure: the result weak classifier Ψ has the lowest weighted error on the learning sample set.

rithm 4. The main differences introduced with AGA (see on Algorithm 4) are as follows:

- The size of the population is fixed to N. The population is partitioned into three classes: best to keep, looser to keep, and looser.

  The best to keep part is of size N_{best}. It represents the classifiers with lowest error. These classifiers will be replaced by the result of the Mutation operation with several mutation types.(see Algorithm 4(line.19))

  The looser to keep part is of size N_{looser}. It represents the classifiers which will be replaced by randomized classifiers. A randomized classifier is a random classifier that is generated by simple modifications to an initial classifier. The result classifier is not completely random.(see Algorithm 4(line.24))

  The looser part is of size $N-(N_{best}+N_{looser})$. It represents the classifiers which will be rejected from the population and replaced by new random ones.(see Algorithm 4(line.29))

- The number of iterations is not fixed to a maximum number of generations G. Otherwise, we introduce the notion of maximum number of iterations without improvement N_{maxgen}. This means that the loop stopped once there are no improvement of Ψ during N_{maxgen} iterations of the weak learner.

Parameter	Description
N	Size of the Population
N_{best}	Number of classifiers to Mutate
N_{looser}	Number of classifiers to Randomize
N_{maxgen}	Max number of iterations without improvement

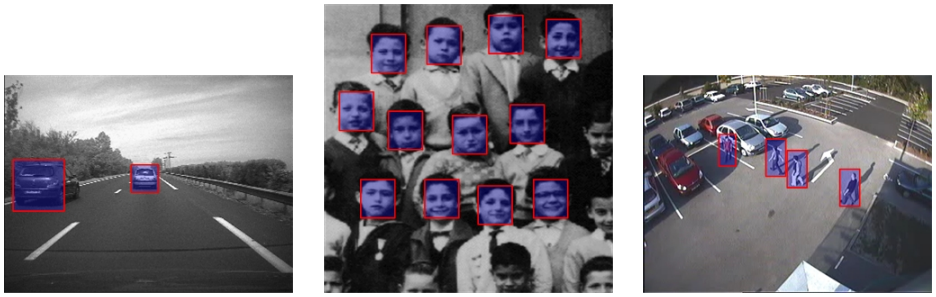
Table 6.1: AGA: parameters description

The AGA algorithm has a set of parameters which are summarized in Table 6.1. We have elaborated an experimental study to find a set of best parameters for this algorithm. Our experimental tests show how the AGA algorithm performs better than the previous GA algorithm.

6.3 Visual domain

Visual domain consists in working with images. The objective is to find visual objects in a given image. A visual object can be a face, a car, a pedestrian or whatever we want. Figure 6.3 shows some examples of visual objects: a visual object has a width and a height, and is characterized by a knowledge generated using a learning procedure.

In this context we consider a sliding window technique for finding visual objects in input images. Sliding window techniques follow a top-down approach. These



(a) (b) (c)

Figure 6.3: Visual object-detection examples

techniques use a detection window of a fixed size and place this detection window over the input image. Next the algorithm determines whether the content of the image inside the window represents an object of interest or not. When it finishes processing the window content, the window slides to another location on the input image and the algorithm again tries to determine whether the content of the detection window represents an object of interest or not. This procedure continues until all image locations are checked. To detect an object of interest at different scales the image is usually scaled down a number of times to form a so-called image pyramid and the detection procedure is repeated.

In this section we will discuss the different elements of a visual detector: section 6.3.1 and section 6.3.2 describe static and dynamic detectors. Section 6.3.3 presents the different visual weak classifiers, from the state of the art as well as from our work. The learning procedure with these weak classifiers is presented in section 6.3.4. Section 6.3.6 describes the test procedure so called finder.

6.3.1 Static detector

A static detector uses spacial information only, and doesn't take into account the temporal information. Thus, the visual object is represented only by an image with dimensions $Height \times Width$. This image sample is called static image sample and is formally presented in Definition 9.

Definition 9 (Static image sample)**.** *A static Image Sample I^s is a visual sample which represents an image of dimensions $Height \times Width$. The static image sample is used as input for static detectors.*

The static detector can be applied to single images, as well as on video streams. It handles the video stream, each image independently of the others. The information on the motion is not used, otherwise it becomes a dynamic detector presented in the next section.

6.3.2 Dynamic detector

In order to use the motion information in a video stream, we introduce the notion of dynamic detector. The dynamic detector can be applied on video streams only. The detector uses the current image I and a buffer of p next images. Thus, the image sample contains image information from multiple images sources as described in Definition 10.

Definition 10 (Dynamic image sample)**.** *A dynamic Image Sample I^d is a visual sample which represents a collection of images $I^d_t, I^d_{t+1} \dots I^d_{t+p}$ with similar dimensions $Height \times Width$. These images correspond to the same window in a video stream. These images distributed on the temporal space contain information on the motion of the visual object. This information will be integrated to the knowledge on the visual object. The dynamic image sample is used as input for dynamic detectors.*

6.3.3 Weak classifiers

A visual weak classifier (visual feature) has a two stage architecture (see Figure 6.4): the first stage does the preprocessing on the input image sample (static or dynamic). The preprocessing result is then used as an input to the classification rule. The classification rule is a simple computation function which decides whether the input corresponds to the object of interest or not. In the following sections we describe

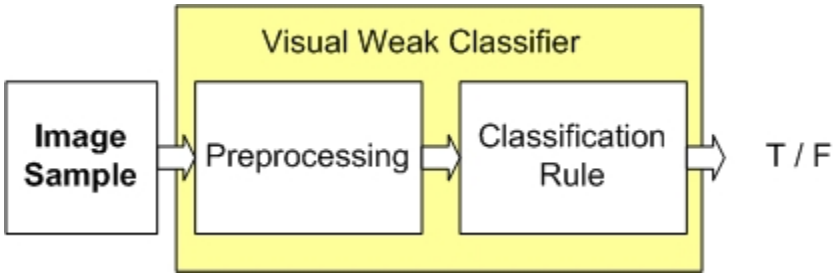


Figure 6.4: Visual weak classifier (Visual Feature) pattern

the most common visual features. We map these features to the architecture pattern described in Figure 6.4. This projection is useful for subsequent complexity analysis and generic implementation as described in section 6.4.

Viola and Jones Rectangular Features

In [VJ01b] Paul Viola and Michael Jones learned a visual detector for face detection using AdaBoost. The learned detector is based on rectangular features (visual weak classifiers) which span a rectangular area of pixels within the detection window, and can be computed rapidly when using a new image representation called the integral image. This section will discuss the rectangular features used by Viola and Jones as shown in Figure 6.5. First we will discuss the classification rules. Then we will

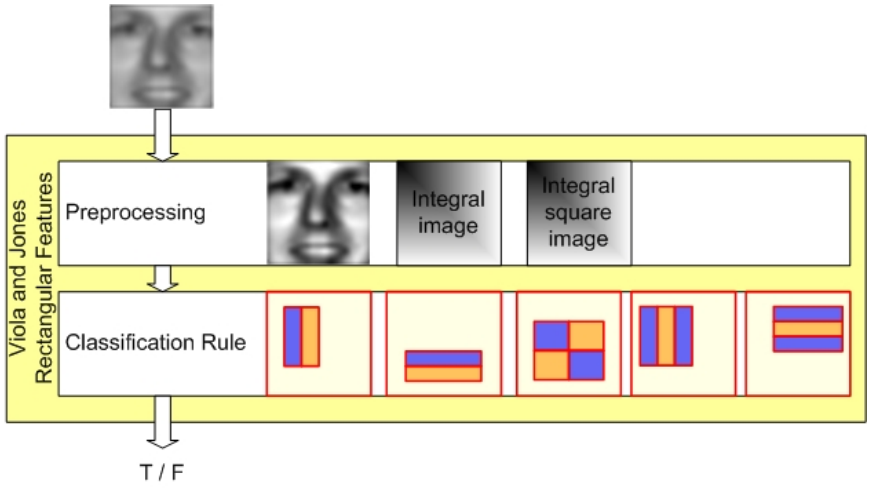
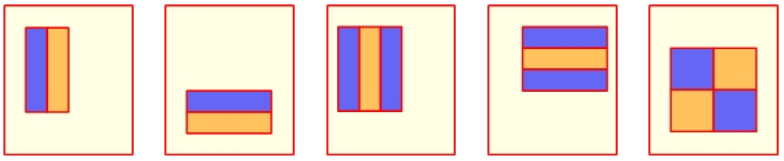


Figure 6.5: Viola and jones rectangular features

describe the preprocessing stage which introduces the integral image. Next, how these features can be computed using the integral image is discussed.



(a) F_{vEdge} (b) F_{hEdge} (c) F_{vLine} (d) F_{hLine} (e) $F_{Diagonal}$

Figure 6.6: The five classification rules used by Viola and Jones placed in the detection window.

1. **Rectangular features**: Figure 6.12 shows the five rectangular features used by Viola and Jones placed in the detection window. To compute a feature, the sum of the pixels in the dark area is subtracted from the sum of the pixels in the light area. Notice that we could also subtract the light area from the dark area, the only difference is the sign of the result. The vertical and horizontal two-rectangle features F_{vEdge} and F_{hEdge} are shown in Figure 6.6(a) and Figure 6.6(b) respectively, and tend to focus on edges. The vertical and horizontal three-rectangle filters F_{vLine} and F_{hLine} are shown in Figure 6.6(c) and Figure 6.6(d) respectively, and tend to focus on lines. The four rectangle feature $F_{Diagonal}$ in Figure 6.6(e) tends to focus on diagonal lines.

2. **Integral image**:Viola and Jones propose a special image representation called the integral image to compute the rectangular filters very rapidly. The integral image is in fact equivalent to the Summed Area Table (SAT) that is used as a texture mapping technique, first presented by Crow in [Cro84]. Viola and Jones

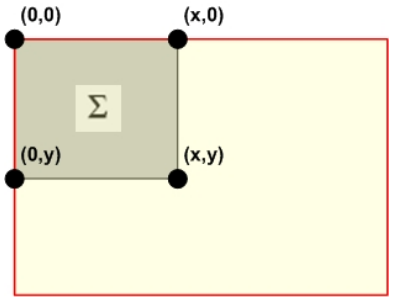


Figure 6.7: Integral image

renamed the SAT to integral image to distinguish between the purpose of use: texture mapping versus image analysis.

The integral image at location (x, y) is defined as the sum of all pixels above and to the left of (x, y) (see Figure 6.7):

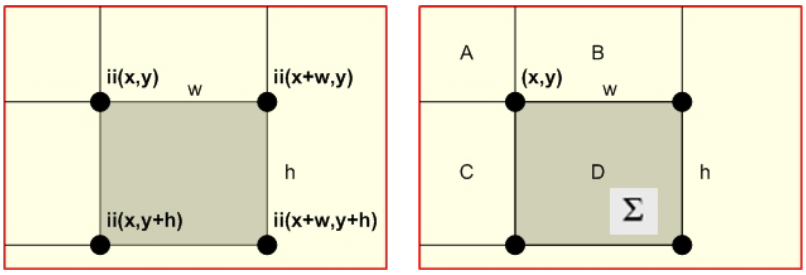
$$ii(x, y) = \sum_{j=0}^{x} \sum_{k=0}^{y} (I(j, k))$$

where $ii(x, y)$ is the integral image value at (x, y), and $I(x, y)$ is the original image value.

3. **Feature computation**: When using integral image representation previously described, any pixel sum of a rectangular region (see Figure 6.8) can be computed with only four lookups, two subtractions and one addition, as shown in the following equation:

$$\sum = (ii(x + w, y + h) + ii(x, y)) - (ii(x + w, y) + ii(x, y + h))$$

We note that this sum is done in constant time regardless of the size of the rectangle region. The five features as proposed by Viola and Jones consist of



(a) (b)

Figure 6.8: Calculation of the pixel sum in a rectangular region using integral image

two or more rectangular regions which need to be added together or subtracted

from each other. Thus, the computation of a feature is reduced to a finite number of pixel sums of rectangular regions. Which can be done easily and in a constant time regardless the size and position of the features.

4. **Image normalization**: Viola and Jones normalized the images to unit variance during training to minimize the effect of different lighting conditions. To normalize the image during detection they post multiply the filter results by the standard deviation σ of the image in the detector window, rather than operating directly on the pixel values. Viola and Jones compute the standard deviation of the pixel values in the detection window by using the following equation:

$$\sigma = \sqrt{\mu^2 - \frac{1}{N}\sum x^2}$$

where μ is the mean, x is the pixel value and N is the total number of pixels inside the detection window. The mean can be computed by using the integral image. To calculate $\sum x^2$ Viola and Jones use a squared integral image, which is an integral image only with squared image values. Computing σ only requires eight lookups and a few instructions.

Control-points features

Figure 6.9: Control-points features

Figure 6.9 shows visual features proposed by Abramson et al [ASG05]. These features work on gray images as in [VJ01b]. Given an image of width W and height H (or a sub-window of a larger image, having these dimensions), we define a *control point* to be an image location in the form $\langle i, j \rangle$, where $0 \leq i < H$ and $0 \leq j < W$. Given an image location z, we denote by $val(z)$ the pixel value in that location.

Control-points feature consists of two sets of control-points, $x_1 \ldots x_n$ and $y_1 \ldots y_m$, where $n, m \leq K$. The choice of the upper bound K influences the performance of the

system in a way which cand be experimented using LibAdaBoost, and in our system we chose $K = 6$. Each feature either works on the original $W \times H$ image, on a half-resolution $\frac{1}{2}W \times \frac{1}{2}H$ image, or on a quarter resolution $\frac{1}{4}W \times \frac{1}{4}H$ image. These two additional scales have to be prepared in advance by downscaling the original image.

To classify a given image, a feature examines the pixel values in the control-points $x_1 \dots x_n$ and $y_1 \dots y_m$ in the relevant image (original, half or quarter). The feature answers "yes" if and only if for every control-point $x \in x_1 \dots x_n$ and every control-point $y \in y_1 \dots y_m$, we have $val(x) > val(y)$. Some examples are given in Figure 6.10.

The control-points feature is testing the relations between pixels. It seeks for a predefined set of pixels in the image in a certain order. If such is not found, it labels negatively. Because it is order-based and not intensity-based, it does not care about *what* is the difference between two pixels' values; all it cares about is the *sign* of that difference. It is therefore insensitive to luminance normalization. In fact, the feature is insensitive to any order-preserving change of the image histogram, including histogram equalization, which is used by Papageorgiou et al [OPS+97]. The

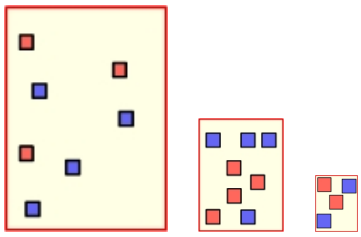


Figure 6.10: Control-Points features

3-resolution method helps to make the feature less sensitive to noise; a single control-point in the lower resolutions can be viewed as a region and thus detects large image components, thus offering an implicit smoothing of the input data.

One can immediately notice that for computing the result of an individual feature, an efficient implementation has not always to check all the $m+n$ control points. The calculation can be stopped once the condition of the feature is broken, and this usually happens much before all the control-points are checked.

Viola and Jones motion-based features

In [VJS03] Viola et al introduced the first dynamic visual feature which mixes patterns of motion and appearance for pedestrian detection. This section will explain these features which are considered as a natural generalization of the rectangular visual features presented above. Figure 6.11 shows a system block representation of the motion-based features. These features operate on a pair of images $[I_t, I_{t+1}]$. Motion is extracted from pairs of images using the following filters:

1. $\Delta = abs(I_t - I_{t+1})$

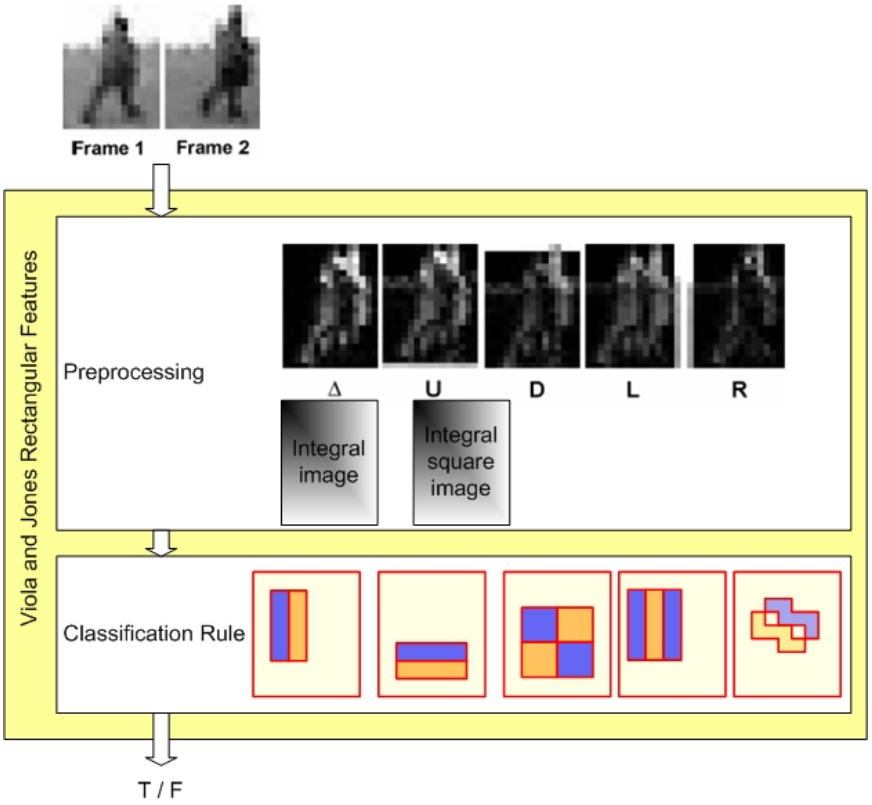


Figure 6.11: Viola and Jones motion-based rectangular features

2. $U = abs(I_t - I_{t+1} \uparrow)$
3. $L = abs(I_t - I_{t+1} \leftarrow)$
4. $R = abs(I_t - I_{t+1} \rightarrow)$
5. $D = abs(I_t - I_{t+1} \downarrow)$

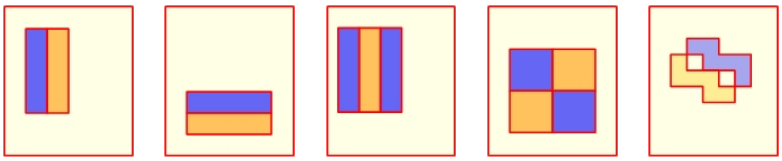


Figure 6.12: Viola and Jones motion-based rectangular features

where I_t and I_{t+1} are images in time, and $\{\uparrow, \leftarrow, \rightarrow, \downarrow\}$ are image shift operators.

One type of filter compares sums of absolute differences between Δ and one of $\{U, L, R, D\}$

$$f_i = r_i(\Delta) - r_i(S)$$

where S is one of $\{U, L, R, D\}$ and $r_i()$ is a single box rectangular sum within the detection window. These filters extract information related to the likelihood that a particular region is moving in a given direction.

The second type of filter compares sums within the same motion image:

$$f_j = \Phi_j(S)$$

where Φ is one of the rectangle filters shown in Figure 6.12. These features measure something closer to motion shear.

Finally, a third type of filter measures the magnitude of motion in one of the motion images:

$$f_k = r_k(S)$$

where S is one of $\{U, L, R, D\}$ and r_k is a single box rectangular sum within the detection window.

We also use appearance filters which are simply rectangle filters that operate on the first input image, It:

$$f_m = \Phi(I_t)$$

The motion filters as well as appearance filters can be evaluated rapidly using the integral image previously described in Section 6.3.3.

A feature, F, is simply a thresholded filter that outputs one of two votes.

$$F_i(I_t, I_{t+1}) = \begin{cases} \alpha & if\ f_i(I_t, \Delta, U, L, R, D) > t_i \\ \beta & otherwise \end{cases} \tag{6.2}$$

where $t_i \in \Re$ is a feature threshold and f_i is one of the motion or appearance filters defined above. The real-valued α and β are computed during AdaBoost learning (as is the filter, filter threshold θ and classifier threshold).

In order to support detection at multiple scales, the image shift operators $\{\uparrow, \leftarrow, \rightarrow, \downarrow\}$ must be defined with respect to the detection scale.

Extended rectangular features

In this section we focus on the question whether more complex visual features based on rectangular features would increase the performance of the classifier. Therefore, we propose two categories of features based on the same pattern as rectangular features. This means that these features use the integral image representation as well as the image normalization using the integral square image representation (see Figure 6.13). The visual filters introduced by these features are as follows:

- **Box-based rectangular features**: These features are based on a basic box which encapsulates another box. The internal box is located in the middle as in F_{box} or at the corner as in $\{F_{UL}, F_{DL}, F_{DR}, F_{UR}\}$.(see Figure 6.14)
- **Grid-based rectangular features**: These features are based on the previous rectangular features F_{vLine}, F_{hLine}, and $F_{Diagonal}$. We introduced lower granularity to these features by decomposing each rectangular region to several smaller regions.(see Figure 6.15)

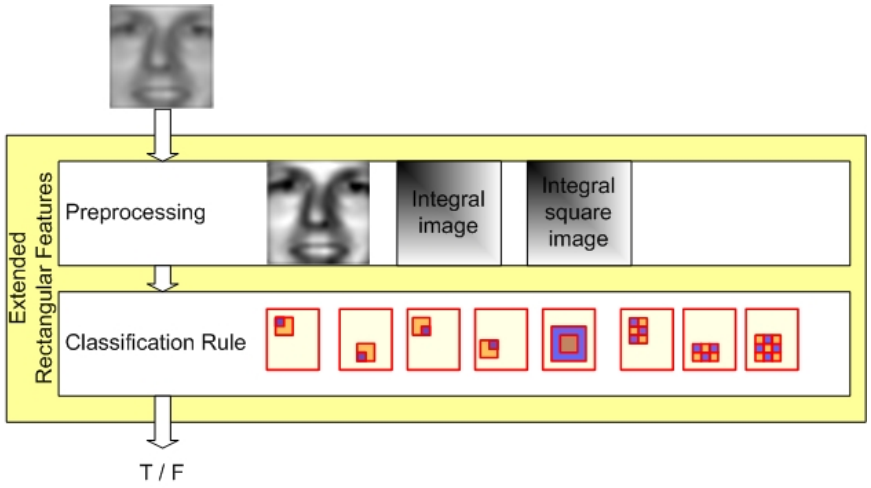
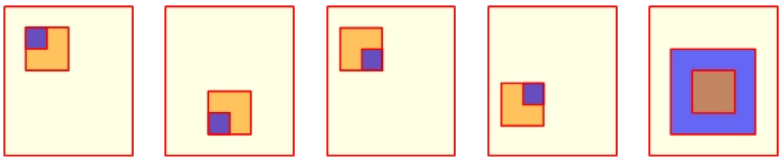


Figure 6.13: Extended rectangular features



(a) F_{UL} (b) F_{DL} (c) F_{DR} (d) F_{UR} (e) F_{box}

Figure 6.14: Classification rules used within extended rectangular features

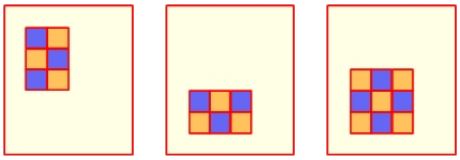
6.3.4 Genetic weak learner interface

The genetic like weak learner described in section 6.2.5 operates on weak classifiers through a genetic interface. The genetic interface provides three operations: randomize, mutate and crossover. In this work we only handled the two first operations randomize and mutate. In this section we will describe the genetic interface for each of the previous visual features.

The visual features are decomposed into two categories: the rectangular features (Viola and Jones rectangular features, motion-based rectangular features and the extended rectangular features) and the control-points features. The first category consists in a set of filters with a generic pattern defined by a rectangular region at a given position in the detection window. This rectangular region is decomposed into multiple rectangular regions. This will generate the different filters. So the genetic interface is based on this rectangular region, so it will be used by all these features in the category. The second category is the control-points features, and it will be described separately.

Rectangular filters

Given an image sample I of width W and height H. Given a rectangular filter F_c^t of category c and type t. where c and t are given in Table 6.2. The randomize operation



(a) F_{VGrid} (b) F_{HGrid} (c) F_{Grid}

Figure 6.15: Extended rectangular features

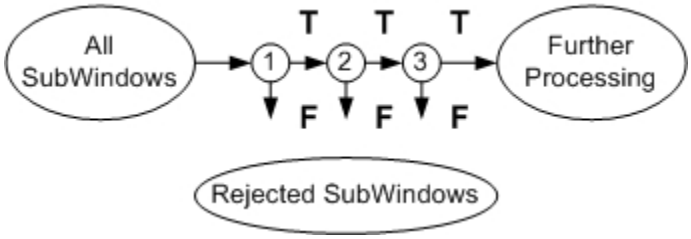


Figure 6.16: Attentional cascade of classifiers

consists in moving the rectangular region randomly within a limited area around the current position. The randomize operation conserves the type and the category of the filter. The mutate operation consists in modifying the dimensions of the rectangular region (scale operation) and to modify the type of the filter without changing the category of the filter.

Control-points filters

Given an image sample I of width W and height H. Given a control-points feature F_r, $r \in \{R_0, R_1, R_2\}$. The randomize operation consists in modifying the position of a red point or a blue point within a limited local area. The randomize conserves the resolution r. The mutation consists in add, remove, move a control-point (red or blue).

6.3.5 Cascade of classifiers

The attentional cascade of classifiers was first introduced by Viola and Jones [VJ01b] as an optimization issue. In a single image, there are very large amount of sliding windows of which the vast majority does not correspond to the object of interest. These sliding windows have to be rejected as soon as possible. The cascade, as shown in Figure 6.16, is composed of a sequence of strong classifiers so called stages (or layers). The first layers contain few weak classifiers and are very fast. The last layers are more complex and contain more weak classifiers. The cascade is learned in such a way: the first layer rejects a large amount of input sliding windows and accepts all positive sliding windows. The next layers are tuned to reduce the amount of false positive rates while still maintaining a high detection rate.

The cascade of classifiers accelerates the detection process. Therefore the single layer classifiers maintain a better accuracy and false detection rate. In this thesis we focus on single layer classifiers and the acceleration of these classifiers using specialized hardware.

6.3.6 Visual finder

Definition 11 (Visual data)**.** *Visual data can be seen as a collection of samples at the abstract level. In the visual domain, the Visual data encapsulates* $I_t, [I_{t+1}, \dots I_{t+p}]$ *where* $p = 0$ *in the case of static detector and* $p > 0$ *in the case of dynamic detector. The Visual data encapsulates also all the subsequent visual data obtained from these images. To obtain the samples (static or dynamic image samples) we use an iterator called search strategy.*

Given a knowledge C_ζ relative to a visual object ζ. Given a visual data (see Definition 11) source π. A visual finder is a component that first builds the visual pyramid for the visual data $\pi(t)$ and does some preprocessing on $\pi(t)$. This preprocessing is required by the classification step. Next it iterates over the sliding windows available through an iterator named search strategy. Each sliding window is classified using C_ζ. Visual finder ends by a postprocessing step which consists in some spatial and temporal filters on the positively classified sliding windows. In the following sections we will detail each of these functional blocks. We start by the generic patterns that could be used to group all these blocks together.

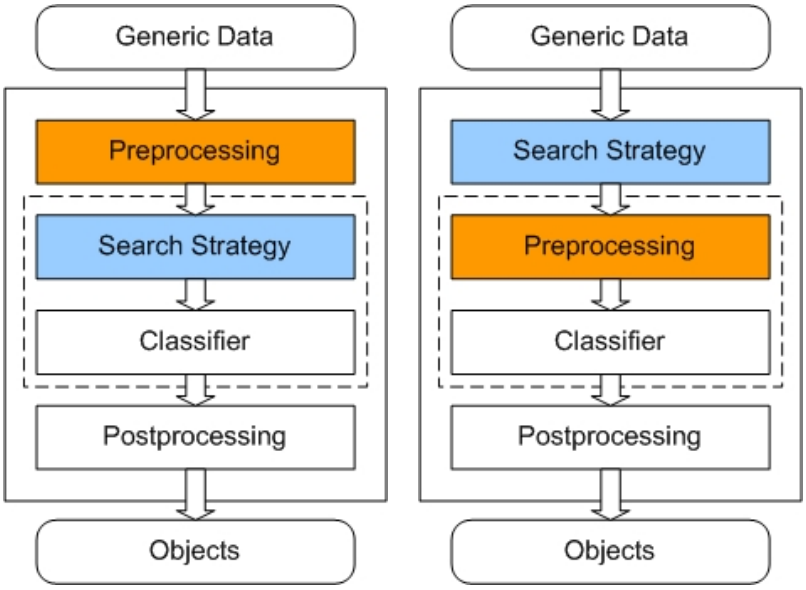
Visual finder patterns

In this section we focus on the question *where to put the preprocessing block relative to the search strategy?* because it influences the performance of the total process in term of memory space and computation time. Figure 6.17(a) shows the first configuration called *Global Preprocessing*. Figure 6.17(b) shows the second configuration called *Local Preprocessing*.

Local preprocessing consists in iterating over the input visual data first. For each sliding window it builds the required preprocessing defined by C_ζ. The sliding window is responsible of the storage of the preprocessing data. This pattern requires less memory size with the cost of more computation time.

Global preprocessing consists in building a global preprocessing data which is shared between all the sliding windows. Each sliding window maintains a reference to the shared preprocessing interface, and a position (s_x, s_y). This pattern requires larger memory space and lower computation time.

These two patterns are constrained by the subsequent architecture. From the functional point of view these two patterns are similar. Thus, in next sections we refer to the visual finder regardless its pattern.

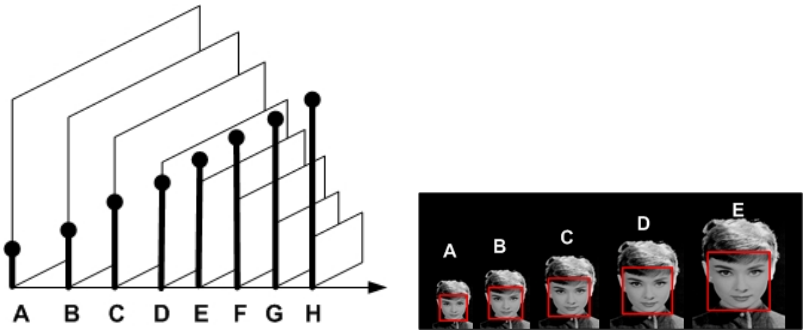


(a) Global Preprocessing (b) Local Preprocessing

Figure 6.17: Generic Finder Patterns

Visual pyramid

In the generic design of the visual detector, we focus on the question *how to find objects at different scales in the same image?* First let us remember that the knowledge C_ζ is specialized on an object ζ with dimensions $h \times w$. The dimensions $h \times w$ represent the dimensions of the smallest object ζ that can be recognized by C_ζ. To detect objects of same category ζ with higher dimensions $h_0 \times w_0$ constrained by $\frac{h}{w} = \frac{h_0}{w_0}$, the classifier C_ζ is applied to the scaled image with factor $\alpha = \frac{h}{h_0}$.



(a) Object to scale association (b) Face example

Figure 6.18: Visual Pyramid: ζ corresponds to 'A', and the largest object found in the image is 'E'.

Given a visual data $\pi(t)$ with embedded image I_t(in case of dynamic detector then we consider same pyramid for all $I_{t+p}, p = 0 \ldots n$). Let $H \times W$ the dimensions

of I_t. The largest object ζ that can be recognized in I_t has as dimensions $H \times W$. Therefore, the objects ζ that could be found in this image have dimensions $h_k \times w_k$ constrained by: $h_k = H \ldots h$, $w_k = W \ldots w$ and $\frac{h_k}{w_k} = \frac{h}{w}$.

We build a set of N images obtained from I_t by a scaling factor α, where $\alpha^N = \min(\frac{h}{H}, \frac{w}{W})$. This set of images is called visual pyramid (see Figure 6.18(a)). The size of the visual pyramid is N. N is calculated by:

$$N = \min(\left\lfloor log_\alpha(\frac{h}{H}) \right\rfloor, \left\lfloor log_\alpha(\frac{w}{W}) \right\rfloor)$$

Table 6.3 represents the size of the pyramid for a quarter-pal image with several values of α. The choise of α depends on the requirements of the application.

Visual finder preprocessing

Given a knowledge C_ζ. Let Ψ be a category of visual features. Let f_Ψ denote a visual feature based on Ψ. For each Ψ, if C_ζ contains at least a visual feature f_Ψ, then the preprocessing block must prepare the required data, as shown in Table 6.4.

Visual finder search strategy

Definition 12 (Visual search strategy)**.** *Given a visual data $\pi(t)$. A search strategy is an iterator on the sliding windows present in $\pi(t)$. The search may cover all the available sliding windows and then we speak in term of* ***full-search****, and otherwise it is called* ***partial-search****.*

We have introduced the notion of search strategy as an abstraction of the scan method presented by Viola and Jones [VJ01b]. We call the previous method the generic visual search. It is fully customizable: this will help us to compare the performance of different visual features without doubts. We can configure the horizontal and vertical steps to zero and then it is similar to full-search strategy as shown in Figure 6.19. Otherwise, Figure 6.19 shows that these parameters decrease dramatically the number of sliding windows in the image. The accuracy of the detection decreases as well. The ideal case is to cover all the sliding windows but this is a heavy task for the traditional processors.

The notion of search strategy is useful, since it can be extended to untraditional search methods. For example in some applications, the objects of interest are located in some regions at different layers in the pyramid. These informations could be measured using statistics on some detection results applied to recorded sequences.

Visual finder postprocessing

Given a knowledge C_ζ and a visual data $\pi(t)$. The detector based on C_ζ is insensitive to small variations in scale and position, usually a large number of detections occur around an object of interest ζ (see Figure 6.20(a)). To incorporate these multiple detections into a single detection, which is more practical, we apply a simple

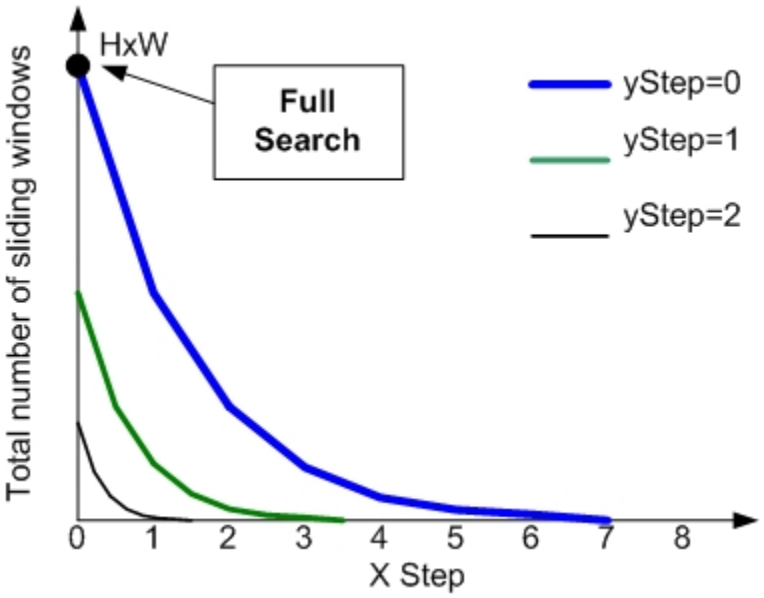
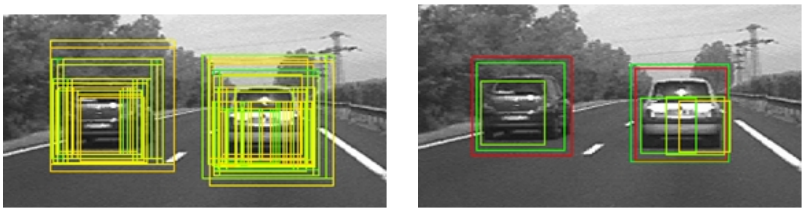


Figure 6.19: Generic Visual Search Strategy

grouping algorithm as in [VJ01b]. This algorithm combines overlapping detection rectangles into a single detection rectangle. Two detections are placed in the same set if their bounding rectangles overlap. For each set the average size and position is determined, resulting in a single detection rectangle per set of overlapping detections (see Figure 6.20(b)).



(a) Before (b) After

Figure 6.20: Visual postprocessing: grouping

6.4 LibAdaBoost: Library for Adaptive Boosting

6.4.1 Introduction

We started developing LibAdaBoost as a unified framework for boosting. This framework implements an extension to the computer vision domain. We have designed the boosting framework using a basic abstraction called *"open machine learning framework"*. Thus, we consider LibAdaBoost as an open machine learning framework which

provides a boosting framework as well as an extension of this framework for the vision domain.

Boosting is for instance used for signal processing (analysis and prediction), image and video processing (face, people, cars ..) or speech processing (speech recognition ...).

Every year, many new algorithms (boosting algorithms, weak learners and weak classifiers) are proposed in various international conferences and journals. It is often difficult for scientists interested in solving a particular task (say visual object detection) to implement them and compare them to their usual tools.

The aim of this section is to present LibAdaBoost, a new machine learning software library available to the scientific community under the Less GPL license, and which implements a boosting framework and a specialized extension for computer vision.

The objective is to ease the comparison between boosting algorithms and simplify the process of extending them through extending weak classifiers, weak learners and learners or even adding new ones.

We implemented LibAdaBoost in C++. We designed LibAdaBoost using well-known patterns. This makes LibAdaBoost easy to take in hand by the majority of the researchers. LibAdaBoost is too large, it consists in more than twenty packages with more than two hundred classes. In this section we focus on the functional description of the different parts of LibAdaBoost. And we give also a sufficient presentation of the global architecture. More advanced technical details are available in [Gho06].

This section is organized as follows: Section 6.4.2 presents the machine learning framework abstraction as exposed in LibAdaBoost. Section 6.4.2 presents the boosting framework built upon the machine learning framework. Section 6.4.2 describes the computer vision extension of the boosting framework. An architecture overview of LibAdaBoost is given in Section 6.4.3, in this section we present both the plugin-based and service oriented approaches. Section 6.4.4 covers the different plugins within LibAdaBoost, and regroups them into four levels: architecture, machine learning framework, boosting framework, and computer vision extension. We compare LibAdaBoost with other available tools in Section 6.4.5; this is followed by a quick conclusion.

6.4.2 LibAdaBoost functional overview

Figure 6.21 shows a high level functional view of LibAdaBoost. It consists in three conceptual levels: machine learning framework, boosting framework and computer vision extension of this boosting framework. The potential users of LibAdaBoost come from two large communities: machine learning community and computer vision community. The machine learning community can improve the first two layers and the computer vision users are more specialized in deploying these algorithms in computer vision domain. In the following we will describe the main concepts that we introduced in each of these frameworks.

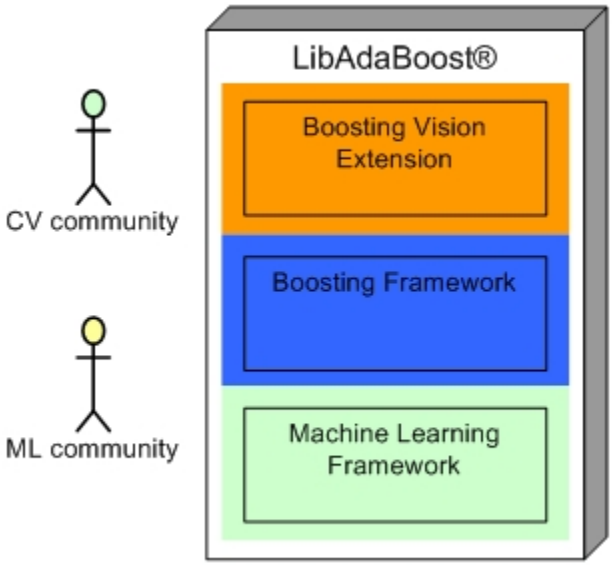


Figure 6.21: LibAdaBoost high level functional diagram

Machine learning framework

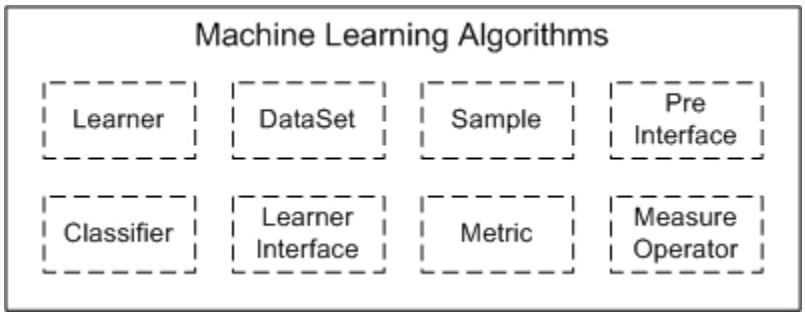


Figure 6.22: Machine learning framework principal concepts

A machine learning framework consists in providing concepts for describing learning algorithms. We focus on statistical machine learning algorithms. These algorithms can be used to build systems able to learn to solve tasks given both a set of samples of that task which were drawn from an unknown probability distribution and with some a priori knowledge of the task. In addition, it must provide means for measuring the expected performance of generalization capabilities. Thus, in this machine learning framework (see Figure 6.22), we provide the following main concepts:

- DataSet, Sample and Pre-Interface: these concepts cover data representation. A Sample represents an elementary data input and a DataSet represents a set of Samples which is used as input for the learning system. The preprocessing

interface represents a way for extending the capabilities of a Sample to be able to communicate with high level concepts, such that the learning system.

- Classifier: This concept represents the knowledge that is used to classify a given Sample. A classifier can be an instance of neural network, a support vector machine, a Hidden Markov Model or a boosted strong classifier.

- Learner and Learner Interface: A learner is used to generate a Classifier according to a given criterion and a given DataSet. It will test its generalization using another DataSet (or the same DataSet).

- Metric and Measure Operator: a metric represents a measure of interest such as classification error, or more advanced as AUC of a ROC curve or others. a Measure operator is the functional block that is responsible of the generation of a given metric.

Boosting framework

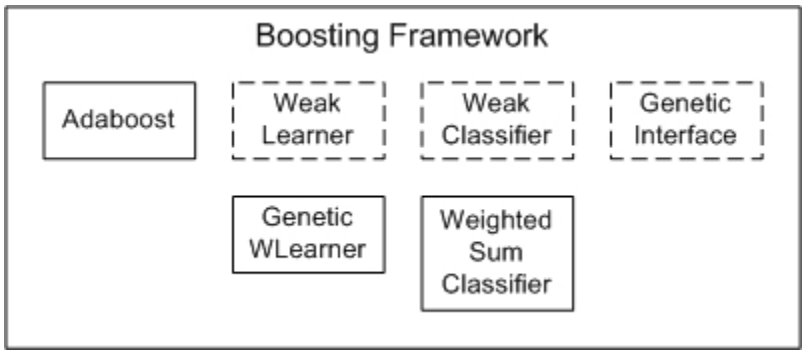


Figure 6.23: Boosting framework principal concepts

A boosting framework as shown in Figure 6.23 consists in few collaborating concepts: the "Boosting algorithm", as AdaBoost [Ref to Shapire and Freund] train several models from a given "Weak Learner" using variations of the original DataSet and then combine the obtained models by a "weighted sum of classifiers". This first procedure is called training procedure. The second procedure in the boosting framework is the test procedure which consists in applying the classifier on sets of samples as input. These sets have specific organization of the samples, thus we use a kind of Finder which once given a classifier and a source of generic data, consists in applying the classifier to each sample present in the generic data. This sample is given by a search strategy implemented in the Finder. the finder has three stages preprocessing, classification and post processing. A non exhaustive list of new introduced concepts is as follows:

- Boosting algorithm (AdaBoost)

- Weak Learner , Genetic Weak Learner and Genetic Interface
- Weak classifier and Weighted sum classifier
- Generic Data
- Finder, Finder preprocessor and Finder post processor

Boosting vision extension

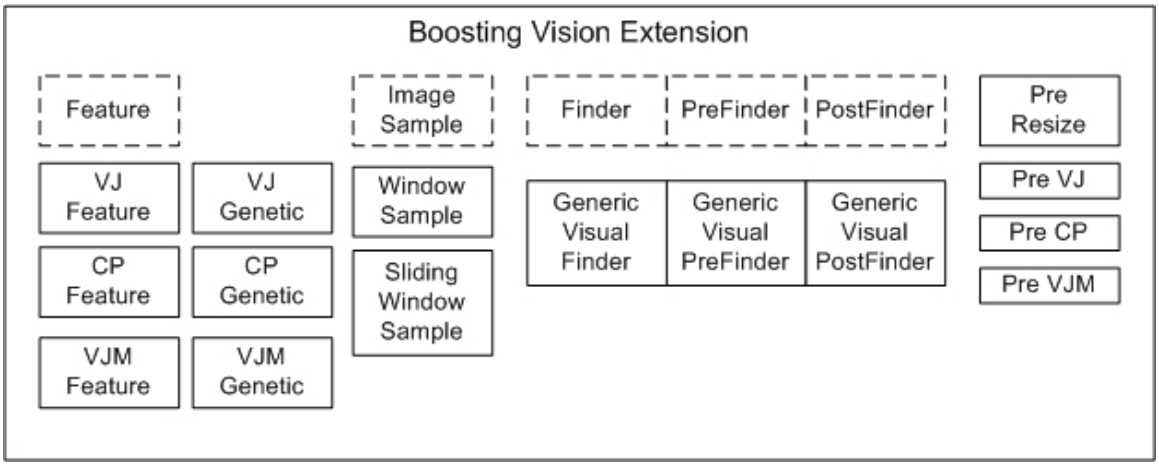


Figure 6.24: Boosting vision extension framework principal concepts

An extension of the boosting framework for the visual domain (see Figure 6.24) consists in extending the notion of sample to support the visual object representation ImageSample used. An image sample can be static (see Definition 9) or dynamic (see Definition 10). The generic Data is extended to Visual Data which represents an image with the pyramid and the set of preprocessing implemented by the different images at the different scales. The weak classifier is extended to visual weak classifier called Feature. In LibAdaBoost we implemented the rectangular features, the control-points features and the VJ motion-based features. The Visual Finder is responsible for finding patterns corresponding to a given object of interest in the input image or video stream. A non exhaustive list of new introduced concepts is as follows:

- Image sample
- Visual Data and Visual Pyramid
- Feature
- Visual Finder, Visual post and pre processors
- VJ, CP and VJM

6.4.3 LibAdaBoost architectural overview

Plugin-based architecture

A design key in LibAdaBoost is the extensibility. We have adopted an architecture which favorates the extensibility and eases the adding or extending algorithms. The architecture has three granularity levels: modules, components and environments. The lowest granularity is the module and the highest granularity is the environment. A module can be accessed in LibAdaBoost if it is registered to a given module manager (similar pattern for components and environments).

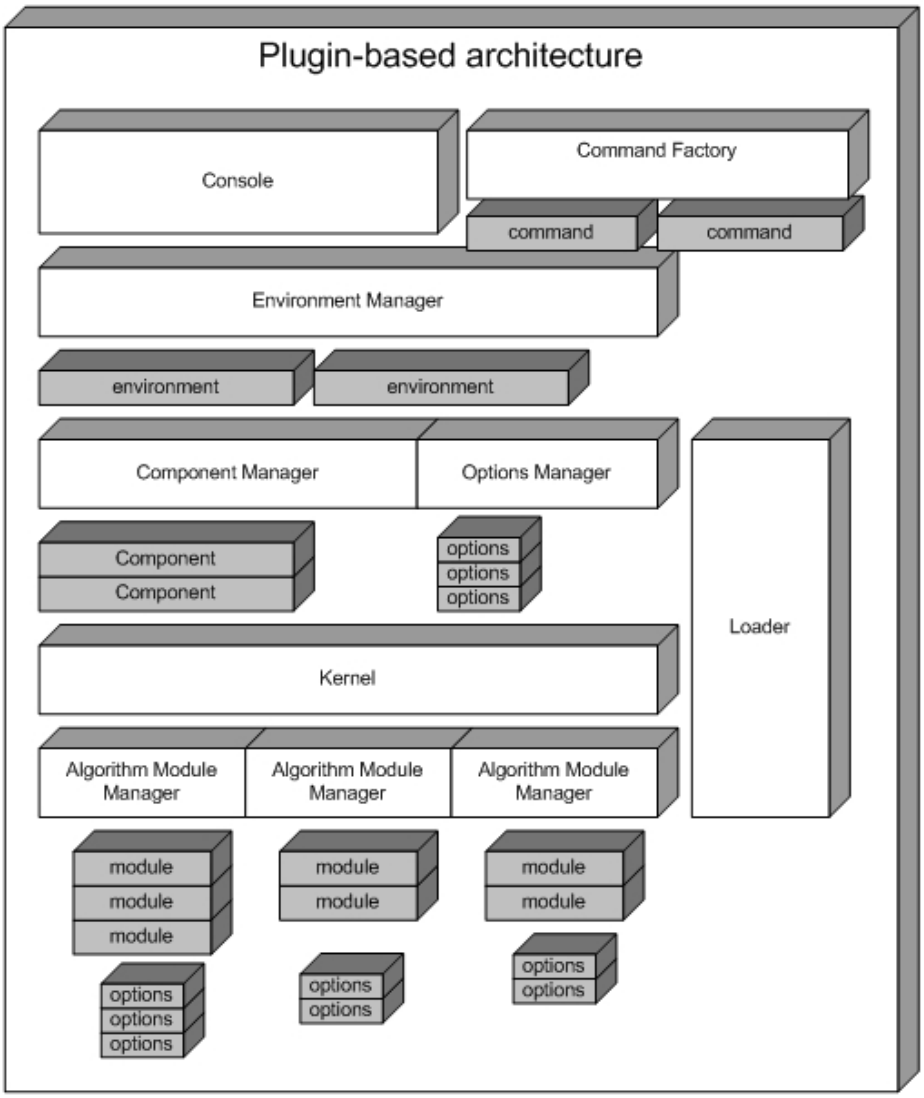


Figure 6.25: Plugin-based architecture objects instantiation

Figure 6.25 shows this plugin-based architecture. We can figure out the following blocks:

- Algorithm module manager, which handles a set of modules implementing the same given interface (example: a learner).
- The kernel is responsible of handling the set of algorithm module managers. It defines the different families of modules available in the system. The component manager handles the set of available components.
- The options manager is used to handle the options required to pass the parameters to these different modules. The environment manager handles the environments objects.
- The console is responsible of the execution of a command handled by the command factory within a given environment.
- The loader is the part responsible of the initialization of the system, it is responsible of loading and doing registration of the different available plugins (modules, components, and environments).

Service oriented tools

LibAdaBoost presents two levels: SDK and toolkit. The toolkit is based on a set of services. These services provide high level functionalities called tasks. We implemented three major services: learning service, validation service and test service.

The service oriented architecture is characterized by the following properties:

- The service is automatically updated to support new functionalities introduced by plugins.
- The service is an extensible part of LibAdaBoost. The developer can easily add more services.
- The different services encode results in a common format.
- A service can be invoked from both: command line and graphical user interface.

Figure 6.26 shows more functional blocks as Launcher and Modeler. Both tools consist in more advanced environments for launching learning tasks, validation tasks and test tasks. We have partially implemented these tools as a web application.

6.4.4 LibAdaBoost content overview

Figure 6.27 presents the fusion of both : the functional analysis and the architectural analysis. We identify four categories of building blocks: the blocks related to the plugin-based architecture, which serve to handle the different modules, and the communication between the different modules, as well within the framework. The blocks related to machine learning framework: these blocks are mostly module managers or component managers and environment managers. The boosting framework consists

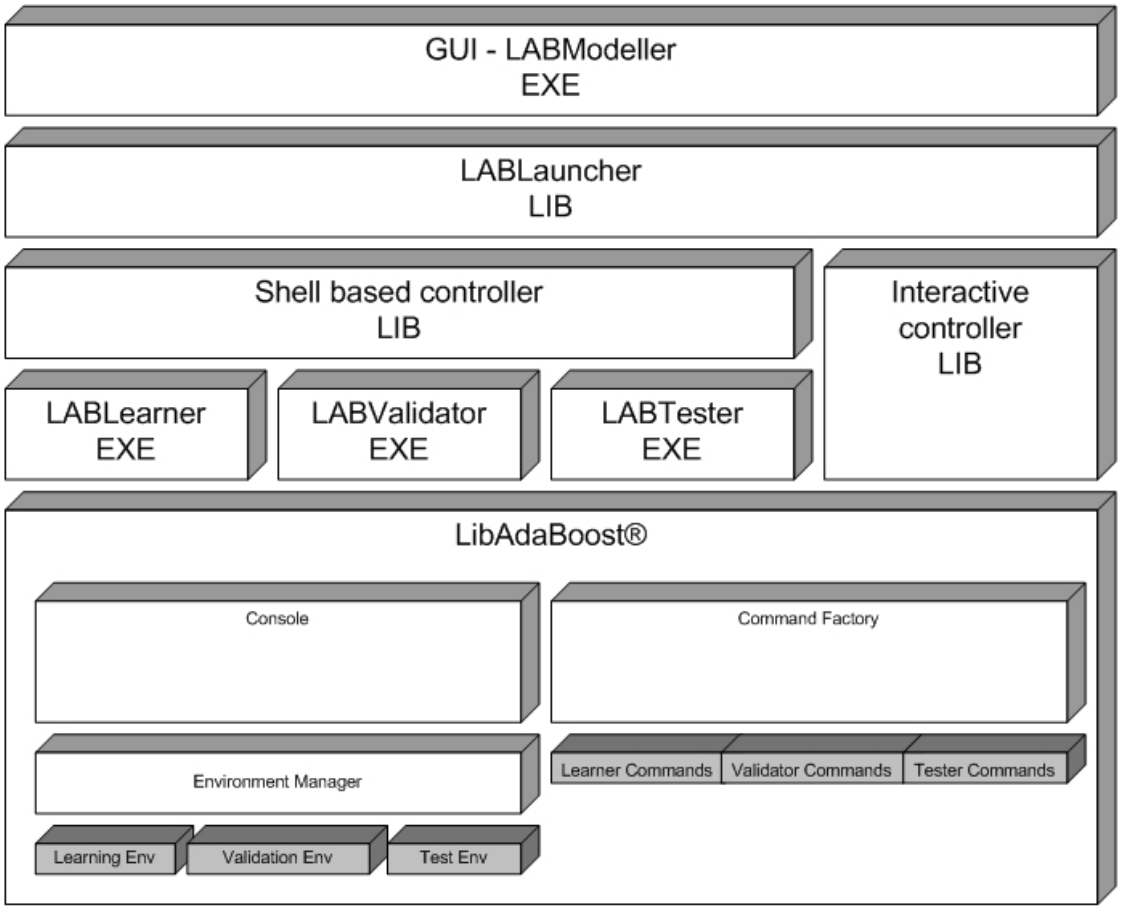


Figure 6.26: LibAdaBoost Service oriented environment

in a set of module managers and modules as plugins. The boosting for vision is a set of plugins implemented upon the boosting framework.

6.4.5 Comparison to previous work

LibAdaBoost is comparable to *Torch* and *Torch Vision* [Put reference of Torch]. Thus we provide more advanced abstraction of the machine learning framework, and the fact that LibAdaBoost is specialized in boosting make it in advanced step compared to torch but in the same time we don't implement the other learning algorithms such as SVM, ... LibAdaBoost is rich in boosting oriented algorithms. Another advantage of LibAdaBoost is the service oriented approach, thus we provide ready to use tools to benefit from the existing algorithms. The plugin based approach make it easy to share and distribute modules.

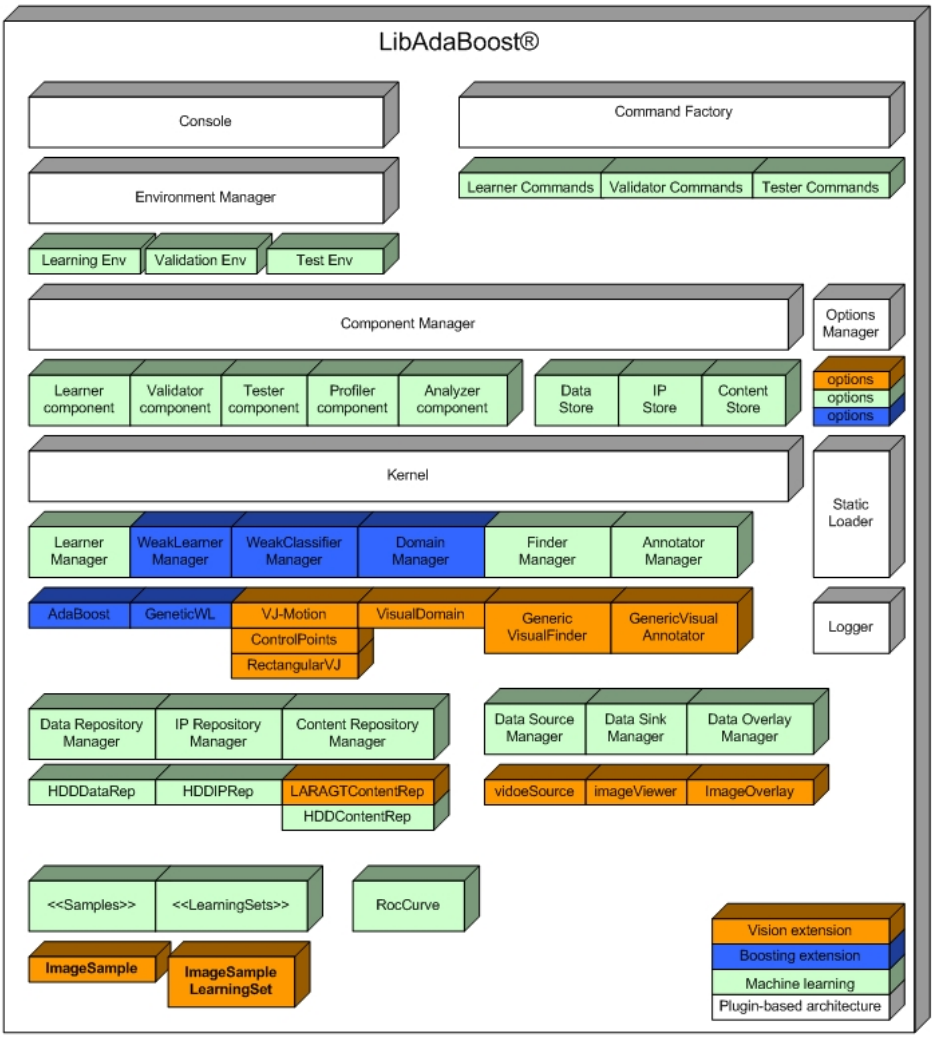


Figure 6.27: LibAdaBoost content overview

6.5 Use cases

In this section we illustrate three use cases for visual object detection: face, car and people. We used LibAdaBoost to solve the task of learning a visual detector for each pattern. We compared several features for this task. We introduced some optimizations to the visual finder in some cases (car detection). We tried to present in each use case a special functionality in LibAdaBoost. These use cases are part of larger projects in the domain of video surveillance and intelligent transportation systems.

6.5.1 Car detection

In this section we illustrate how to use LibAdaBoost for learning a visual detector specialized in cars. The car detection is an important task within a wide application area related to intelligent transportation systems. Vision systems based on frontal cameras, embedded on a vehicle, provide realtime video streams. This video stream is analyzed with special algorithms in the aim to detect cars, people...

We use boosting for learning a special detector specialized in cars. The learning task starts by collecting a set of car samples (1224 samples) and non car samples (2066 samples). These samples are labeled positive and negative respectively. Some examples of these samples are shown in Figure 6.28. A car sample consists in a rear view of a car. The shape of the car is labeled using a bounding box manually selected from the training dataset. The basic size of a car is 48x48 pixels in RGB space.

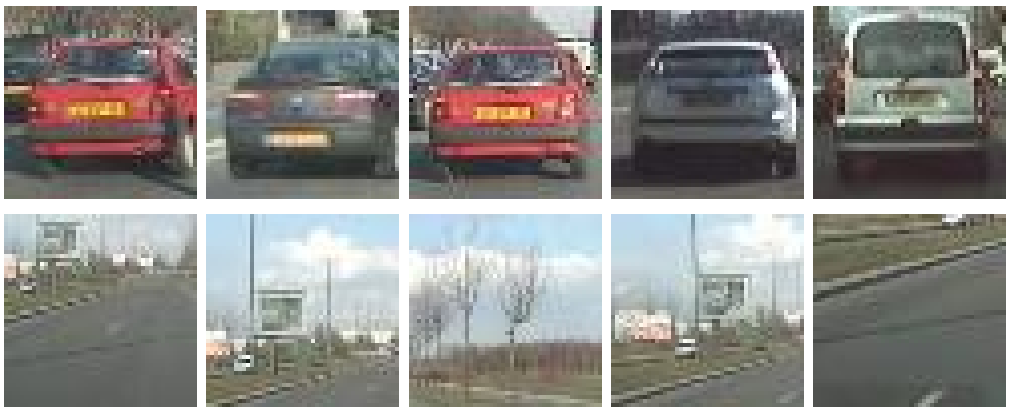
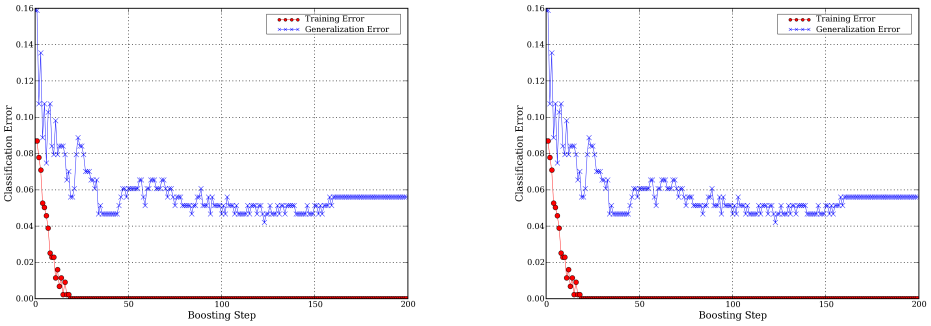


Figure 6.28: Car examples from the learning set: upper line for the positive samples and lower line for the negative examples

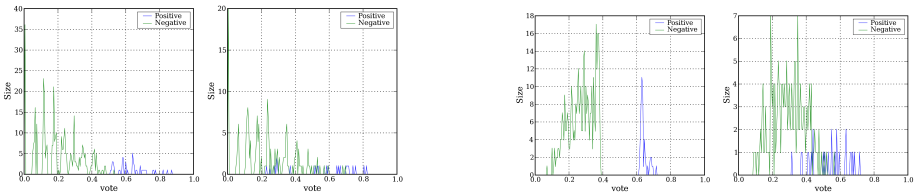


(a) Rectangular Features (b) Control Points

Figure 6.29: Car detection: training error and generalization error

Figure 6.29 shows the evolution of both training and generalization errors for two learning tasks: (a) shows the evolution of the training using rectangular features and

(b) shows the evolution of the training using control points.



(a) Step 15 (b) Step 160

Figure 6.30: Car detection: Voting histogram for training and validation datasets

LibAdaBoost offers a log utility which permits to spy the evolution of the voting values distribution using histograms. Figure 6.30 shows the status of the voting histogram at two moments: the first view on step 15 where we find that the positive and negative histograms are mixed. The second view takes place on step 160. It shows that the histograms are separated for the training dataset and still have a small mixed part for the generalization dataset.

Finally, Figure 6.31 shows the validation step of the result knowledge: the ROC curve is plotted as well as the precision/recall curve. The accuracy variation according to the voting threshold shows that the standard 0.5 value is the best for this detector. The validation result of the detector on the validation dataset is shown as a histogram distribution of the votes which shows a relatively small error zone.

6.5.2 Face detection

In this section we illustrate how to use LibAdaBoost for learning a visual detector specialized in faces. The face detection is an important task within a wide application area related to mobile and video surveillance domains.

We use boosting for learning a special detector specialized in faces. The learning task starts by collecting a set of face samples (4870) and non face samples (14860). These samples are labeled positive and negative respectively. Some examples of these samples are shown in Figure 6.32. A face sample consists in a frontal view of a face. The shape of the face is labeled using a bounding box manually selected from the training dataset. The basic size of a face is 24x24 pixels in RGB space.

Figure 6.33 shows the evolution of both training and generalization errors for two learning tasks: (a) shows the evolution of the training using rectangular features and (b) shows the evolution of the training using control points.

LibAdaBoost offers a log utility which permits to spy the evolution of the voting values distribution using histograms. Figure 6.34 shows the status of the voting histogram at two moments: the first view on step 15 where we find that the positive and negative histograms are mixed. The second view takes place on step 160. It shows that the histograms are separated for the training dataset and still have a small mixed part for the generalization dataset.

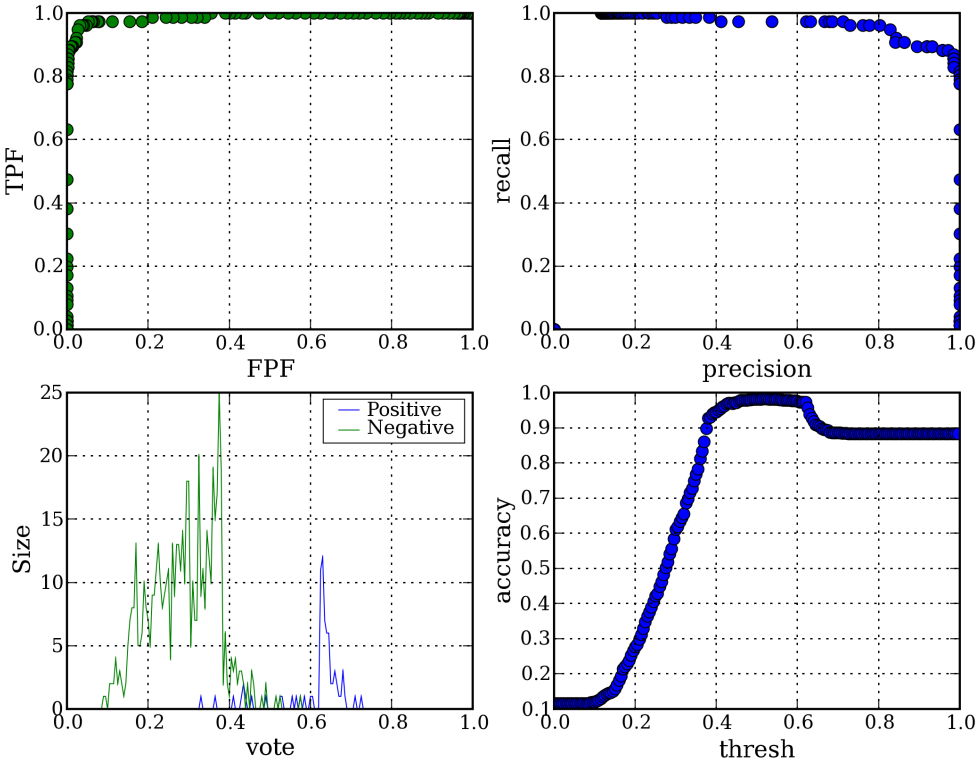


Figure 6.31: Car detection: LibAdaBoost validation result

Finally, Figure 6.35 shows the validation step of the result knowledge: the ROC curve is plotted as well as the precision/recall curve. The accuracy variation according to the voting threshold shows that the standard 0.5 value is the best for this detector. The validation result of the detector on the validation dataset is shown as a histogram distribution of the votes which shows a relatively small error zone.

6.5.3 People detection

In this section we illustrate how to use LibAdaBoost for learning a visual detector specialized in people. The people detection is an important task within a wide application area related to intelligent video surveillance systems. Real time video streams are analyzed with special algorithms in the aim to detect people and then track them and do further post processing and high level analysis.

We use boosting for learning a special detector specialized in people. The learning task starts by collecting a set of people samples and non people samples. These samples are labeled positive and negative respectively. Some examples of these samples are shown in Figure 6.36. A people sample consists in a complete view of a person. The shape of the person is labeled using a bounding box manually selected from the

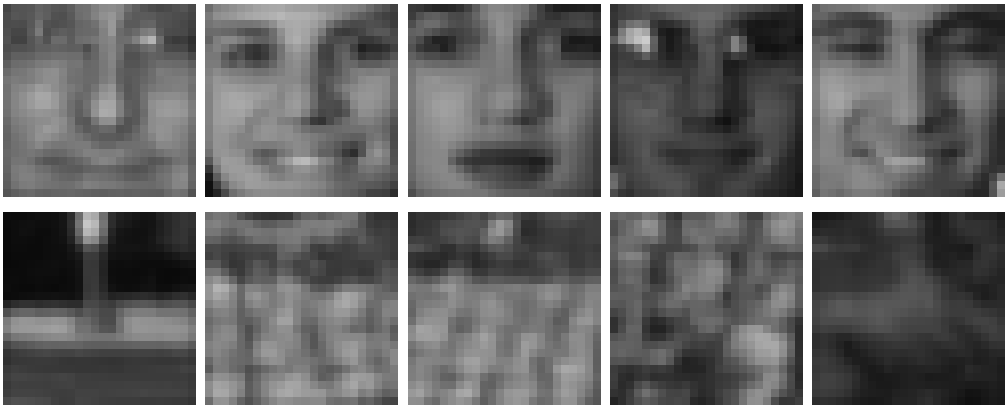
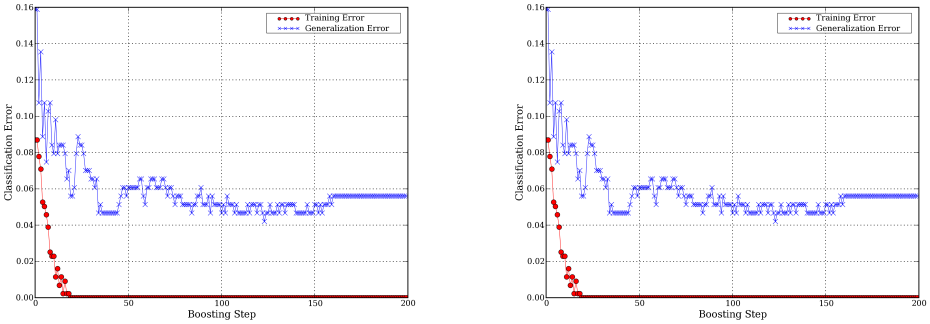


Figure 6.32: Face examples from the learning set: upper line for the positive samples and lower line for the negative examples



(a) Rectangular Features (b) Control Points

Figure 6.33: Face detection: earning and generalization error

training dataset. The basic size of a person is 36x48 pixels in RGB space.

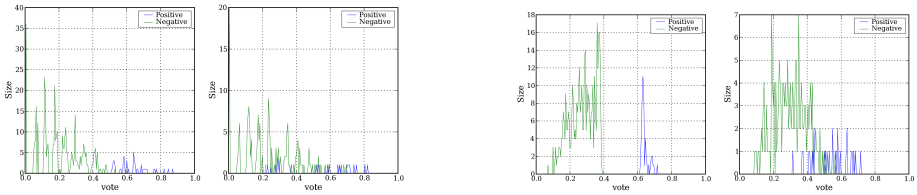
People knowledge depends on the camera installation, the view angle is closely correlated to the knowledge. Thus for each camera we dispose of a special learning process. This is a disadvantage of the current features (Control Points and rectangular features). This can be resolved by motion based features which is not used in this present use case since some technical bugs still have to be fixed.

We present as in the previous sections the evolution of the error for training as well as for the generalization step. This is shown in Figure 6.37.

6.6 Conclusion

In this chapter we presented a new framework for machine learning. We built a boosting framework based on this framework. We exposed the boosting framework to the computer vision domain.

We made this framework public under the LGPL license. To the best of our knowl-



(a) Step 15 (b) Step 160

Figure 6.34: Face detection: Voting histogram for learning and validation datasets

edge this is a new framework which will serve as a reference for the implementation and comparison of various weak classifiers and learning algorithms.

We showed some examples of how using LibAdaBoost for the generation of knowledge. We did not go deeply inside the technical details. These details are explained in tutorials and programming documentation.

The framework is operational and is ready to be integrated in larger projects as a tool for boosting based detectors.

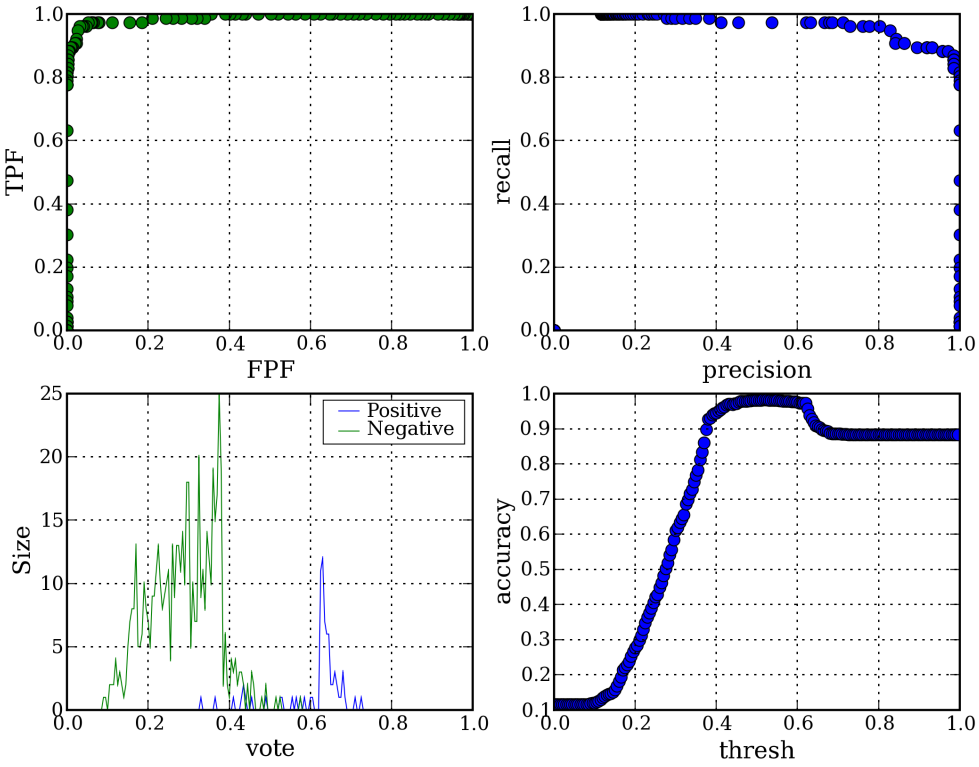


Figure 6.35: Face detection: LibAdaBoost validation result

Category	Type
Viola and Jones Rectangular Feature	F_{vEdge}, F_{hEdge}, F_{vLine}, F_{hLine}, $F_{Diagonal}$
Viola and Jones Motion-based features	motion: f_j, f_k appearance: f_m mixture: f_i
Extended Rectangular features	F_{UL}, F_{DL}, F_{DR}, F_{UR}, F_{box} F_{VGrid}, F_{HGrid}, F_{Grid}

Table 6.2: Category and types association

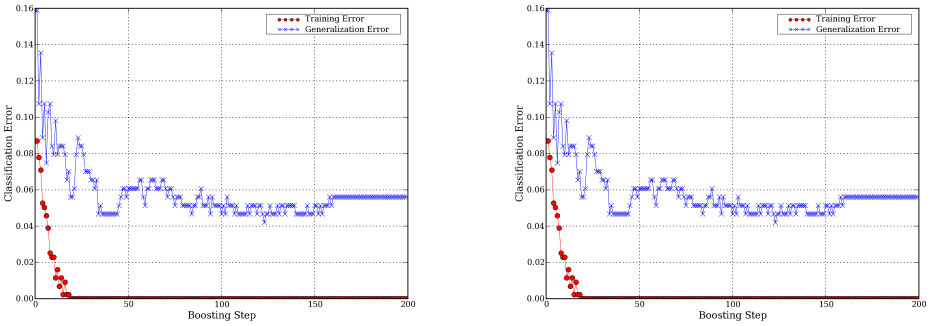
α	N
0.95	48
0.90	23
0.85	15
0.8	11
0.75	8
0.70	6
0.50	3
0.25	1

Table 6.3: Pyramid size for different values of α, image dimensions 384×288, ζ size 24×24

Visual feature	Required preprocessing
Viola and Jones rectangular features	$II_{I_t}, II_{I_t^2}$
Control-points	R_0, R_1, R_2
Motion-based rectangular features	$II_{I_t}, II_{I_t^2}$
	$\Delta, II_{\Delta}, II_{\Delta^2}$
	U, II_U, II_{U^2}
	D, II_D, II_{D^2}
	L, II_L, II_{L^2}
	R, II_R, II_{R^2}

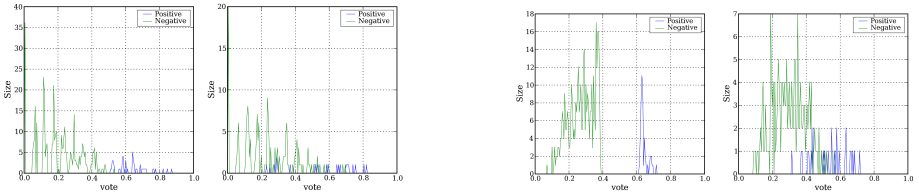
Table 6.4: Required preprocessing for each visual feature

Figure 6.36: People examples from the learning set: upper line for the positive samples and lower line for the negative examples



(a) Rectangular Features (b) Control Points

Figure 6.37: People detection: learning and generalization error



(a) Step 15 (b) Step 160

Figure 6.38: People detection: Voting histogram for learning and validation datasets

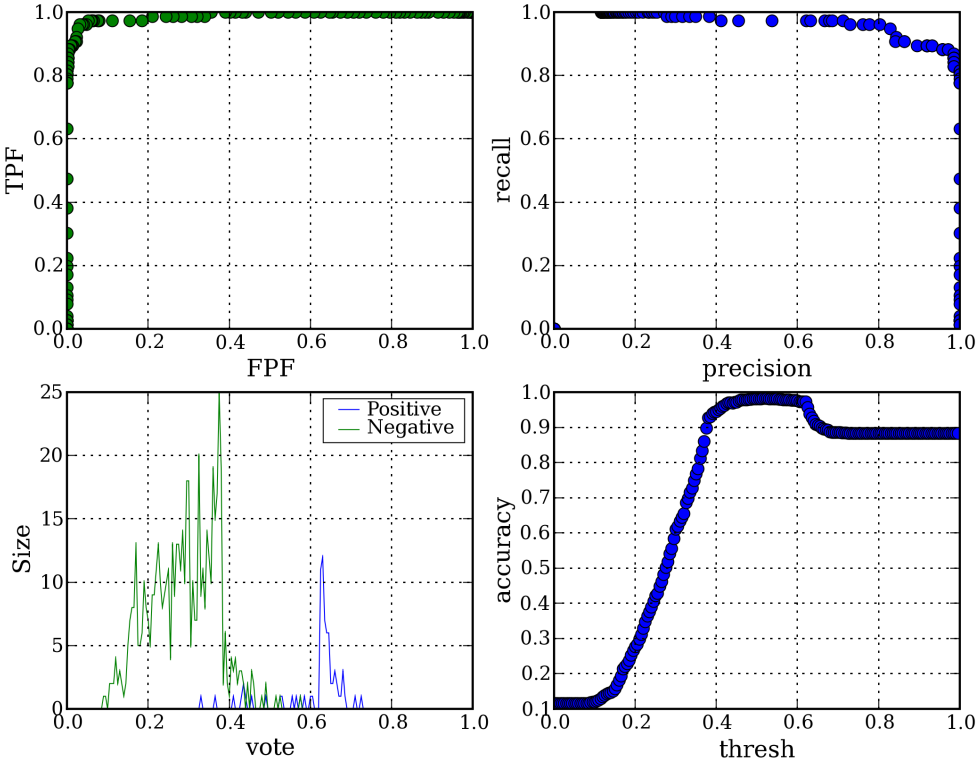


Figure 6.39: People detection: LibAdaBoost validation result

Chapter 7

Object tracking

Contents

The aim of object tracking is to establish a correspondence between objects or object parts in consecutive frames and to extract temporal information about objects such as trajectory, posture, speed and direction. Tracking detected objects frame by frame in video is a significant and difficult task. It is a crucial part of our video surveillance systems since without object tracking, the system could not extract cohesive temporal information about objects and higher level behavior analysis steps would not be possible. On the other hand, inaccurate foreground object segmentation due to shadows, reflectance and occlusions makes tracking a difficult research problem.

Our tracking method is an improved and extended version of the state of art color-based sequential Monte Carlo tracking method proposed in [JM02].

7.1 Initialization

In order to perform the tracking of objects, it is necessary to know where they are initially. We have three possibilities to consider: either manual initialization, semiautomatic initialization, or automatic initialization. For complicated models or objects with a constantly moving background, manual initialization is often preferred. However, even with a moving background, if the background is a uniformly colored region, then semi-automatic initialization can be implemented. Automatic initialization is

possible in many circumstances, such as smart environments, surveillance, sports tracking with a fixed background or other situations with a fixed camera and a fixed background by using the background model and detect targets based on a large intensity change in the scene and the distribution of color on the appearance of an object.

7.2 Sequential Monte Carlo tracking

Sequential Monte Carlo methods have become popular and already been applied to numerous problems in time series analysis, econometrics and object tracking.In non-Gaussian, hidden Markov (or state-space) models (HMM), the state sequence $\{x_t; t^{\in}N\}$,$x_t \in R^{n_x}$, is assumed to be an unobserved (hidden) Markov process with initial distribution $p(x_0)$ and transition distribution $p(x_t/p_(x_{t-1})$, where n_x is the dimension of the state vector. The observations $\{y_t; t^{\in}N^*\}$,$y_t \in R^{n_y}$ are conditionally independent given the process $\{x_t; t^{\in}N\}$ with marginal distribution $p(y_t/x_t)$, where n_y is the dimension of the observation vector. We denote the state vectors and observation vectors up to time t by $x_{0:t} \equiv \{x_0...x_t\}$ and similarly for $y_{0:t}$. Given the model, we can obtain the sequence of filtering density to be tracked by the recursive Bayesian estimation:

$$p(x_t/y_{0:t}) = \frac{p(y_t/x_t)p(x_t/y_{0:t-1})}{p(y_t/y_{0:t-1})} \tag{7.1}$$

where $p(y_t/x_t)$ is likelihood in terms of the observation model, $p(x_t/y_{0:t-1})$ is a prior the propagates part state to future and $p(y_t/y_{0:t-1})$ is evidence. The Kalman filter can handle Eq (7.1) Analytically if the model is based on the linear Gaussian state space. However, in the case of visual tracking, the likelihood is non-linear and often multi-modal with respect to the hidden states. As a result, the Kalman filter and its approximation work poorly for our case.

With sequential Monte Carlo techniques, we can approximate the posterior $p(x_t/y_{0:t-1})$ by a finite set of M particles (or samples), $\{x_t^i\}_{i=1...M}$. In order to generate samples from $p(x_t/y_{0:t-1})$, we propagate samples based on an appropriate proposal transition function $f(x_t/x_{t-1}, y_t)$. We set $f(x_t/x_{t-1}, y_t) = p(x_t/x_{t-1})$, which is the bootstrap filter. We denote $\{x_t^i\}_{i=1...M}$ as fair samples from the filtering distribution at time t, then the new particles, denoted by $\tilde{x}_{t+1}^i$, have the following association with the importance weights:

$$\pi_{t+1}^i \propto \frac{p(_{t+1}/\tilde{x}_{t+1}^i p(\tilde{x}_{t+1}^i/x_t^i}{f(\tilde{x}t+1^i/x_t^i, y_{t+1})} \tag{7.2}$$

where $\sum_{i=1}^{M} \pi_{t+1}^i = 1$. We resample these particles with their corresponding importance weights to generate a set of fair samples $\{x_{t+1}^i\}_{i=1...M}$ from the posterior $p(x_t/y_{0:t})$. With the discrete approximation of $p(x_t/y_{0:t})$, the tracker output is obtained by the Monte Carlo approximation of the expectation:

$$\hat{x}_t \equiv E(x_t/y_t) \tag{7.3}$$

where $E(x_t/y_t) = \frac{1}{M}\sum_{i=1}^{M} x_t^i$.

7.3 State dynamics

We formulate the state to describe a region of interest to be tracked. We assume that the shape, size, and position of the region are known a priori and define a rectangular window W. The shape of the region could also be an ellipse or any other appropriate shapes to be described, which depends mostly on what kind of object to track. In our case,we use a rectangle . The state consists of the location and the scale of the window W.

The state dynamics varies and depends on the type of motion to deal with. Due to the constant, yet often random, nature of objects motion, we choose the second-order auto-regressive dynamics to be the best approximating their motion as in [JM02]. If we define the state at time t as a vector $x_t = (l_t^T, l_{t-1}^T, s_t, s_{t-1})$, where ^{T} denotes the transpose, $l_t = (x, y)^T$ is the position of the window W at time t in the image coordinate, and s_t is the scale of W at time t. We apply the following state dynamics:

$$x_{t+1} = Ax_t + Bx_{t-1} + Cv_t, v_t \equiv N(0, \Sigma). \tag{7.4}$$

Matrices A,B,C and Σ control the effect of the dynamics. In our experiments, we define those matrices in ad-hoc way by assuming the constant velocity on object's motion.

7.4 Color distribution Model

This section explains how we incorporate the global nature of color in visual perception into our sequential Monte Carlo framework. We follow the implementation of HSV color histograms used in [JM02], and extend it to our adaptive color model.

We use histograming techniques in the Hue-Saturation-Value (HSV) color space for our color model. Since HSV decouples the intensity (i.e., Value) from color (i.e., Hue and Saturation), it becomes reasonably insensitive to illumination effects. An HSV histogram is composed of $N = N_h N_s + N_v$ bins and we denote $b_t(k) \in 1, ...N$ as the bin index associated with the color vector $y_t(k)$ at a pixel location k at time t. As it is pointed out in [JM02] that the pixels with low saturation and value are not useful to be included in HSV histogram, we populate the HSV histogram without those pixels with low saturation and value.

If we define the candidate region in which we formulate the HSV histogram as $R(x_t) \equiv l_t + s_t W$, then a kernel density estimate $Q(x_t) \equiv q(n; x_t)_{n=1,...,N}$ of the color

distribution at time t is given by :

$$q(n; x_t) = \eta \sum_{k \in R(x_t)} \delta[b_t(k) - n] \tag{7.5}$$

where δ is the Kronecker delta function, η is a normalizing constant which ensures $\sum_{n=1}^{N} q(n; x_t) = 1$, and a location k could be any pixel location within R(xt). Eq(7.5) defines $q(n; x_t)$ as a probability of a color bin n at time t.

If we denote $Q^* = q_*(n; x_0)_{n=1,\ldots,N}$ as the reference color model and $Q(x_t)$ as a candidate color model, then we need to measure the data likelihood (i.e., similarity) between Q_* and $Q(x_t)$. As in [JM02], we apply the Bhattacharyya similarity coefficient to define a distance ζ on HSV histograms and its formulation given by:

$$\zeta[Q^*, Q(x_t)] = \left[1 - \sum_{n=1}^{N} \sqrt{q^*(n; x_0) q(n; x_t)}\right]^{\frac{1}{2}} \tag{7.6}$$

Once we obtain a distance ζ on the HSV color histograms, we use the following likelihood distribution given by:

$$p(y_t/x_t) \propto \exp^{-\lambda \zeta^2[Q^*, Q(x_t)]} \tag{7.7}$$

where $\lambda = 20$. λ is determined based on our experiments. Also, we set the size of bins N_h, N_s, and N_v as 10.

7.5 Results

This section presents the tracking results by our tracker for a single object. Our tracker tracks a target for many frames if a good initialization were made. The following figures show the result.

These figures show the robustness of our tracker under a severe illumination change with a similar object closely, and the robustness in a cluttered scenes and partial occlusion, the tracker never loses the target.

Figure 7.1 shows illumination insensitive performance of our tracker which does not lose the target even if there is a drastic illumination change due to camera flashes. The same figure shows that the second-order AR dynamics tends to fail to approximate the target's velocity and our tracker may not locate the object exactly, when the object moves by a small amount.

Figure 7.1 and Figure 7.2 show the successful performance of our tracker in a cluttered scene with similar objects nearby (frame 390). These figures also show that the second order AR dynamics makes our tracker robust in a scene when targets make a sudden direction change.

Our Tracker fails when there are identical objects such as players in the same team nearby the target, for the tracker does not know the association between two identical objects without any prior information. This is one open problem that we need to deal with in the future.

7.6 Incorporating Adaboost in the Tracker

This method is introduced in [Ka04]. They adopt the cascaded Adaboost algorithm of Viola and Jones [VJ01a], originally developed for detecting faces. The Adaboost results could be improved if we considered the motion models of objects. In particular, by considering plausible motions, the number of false positives could be reduced. For this reason, we incorporated Adaboost in the proposal mechanism of our tracker. For this the expression for the proposal distribution will be given by the following mixture.

$$Q_b^*(x_t/x_{0:t-1}, y_t) = \alpha Q_{ada}(x_t/x_{t-1}, y_t) + (1-\alpha)p(x_t/x_{t-1}) \tag{7.8}$$

where Q_{ada} is a Gaussian distribution that we discuss in the subsequent paragraph. The parameter α can be set dynamically without affecting the convergence of the particle filter (it is only a parameter of the proposal distribution and therefore its in hence is corrected in the calculation of the importance weights).

Note that the Adaboost proposal mechanism depends on the current observation y_t. It is, therefore, robust to peaked likelihoods. Still, there is another critical issue to be discussed: determining how close two different proposal distributions need to be for creating their mixture proposal. We can always apply the mixture proposal when Q_{ada} is overlaid on a transition distribution modeled by autoregressive state dynamics. However, if these two different distributions are not overlapped, there is a distance between the mean of these distributions.

If a Monte Carlo estimation of a mixture component by a mixture particle filter overlaps with the nearest cluster given by the Adaboost detection algorithm, we sample from the mixture proposal distribution. If there is no overlap between the Monte Carlo estimation of a mixture particle filter for each mixture component and cluster given by the Adaboost detection, then we set $\alpha = 0$, so that our proposal distribution takes only a transition distribution of a mixture particle filter.

7.6.1 Experimental results

We have used for our experiments an implementation in Matlab and initialized our tracker to track three targets. Figure 7.3 shows the results. In this figure, it is important to note that no matter how many objects are in the scene, the mixture representation of the tracker is not affected and successfully adapts to the change. For objects coming into the scene, Adaboost quickly detects a new object in the scene within a short time sequence of only two frames. Then the tracker immediately assigns particles to an object and starts tracking it.

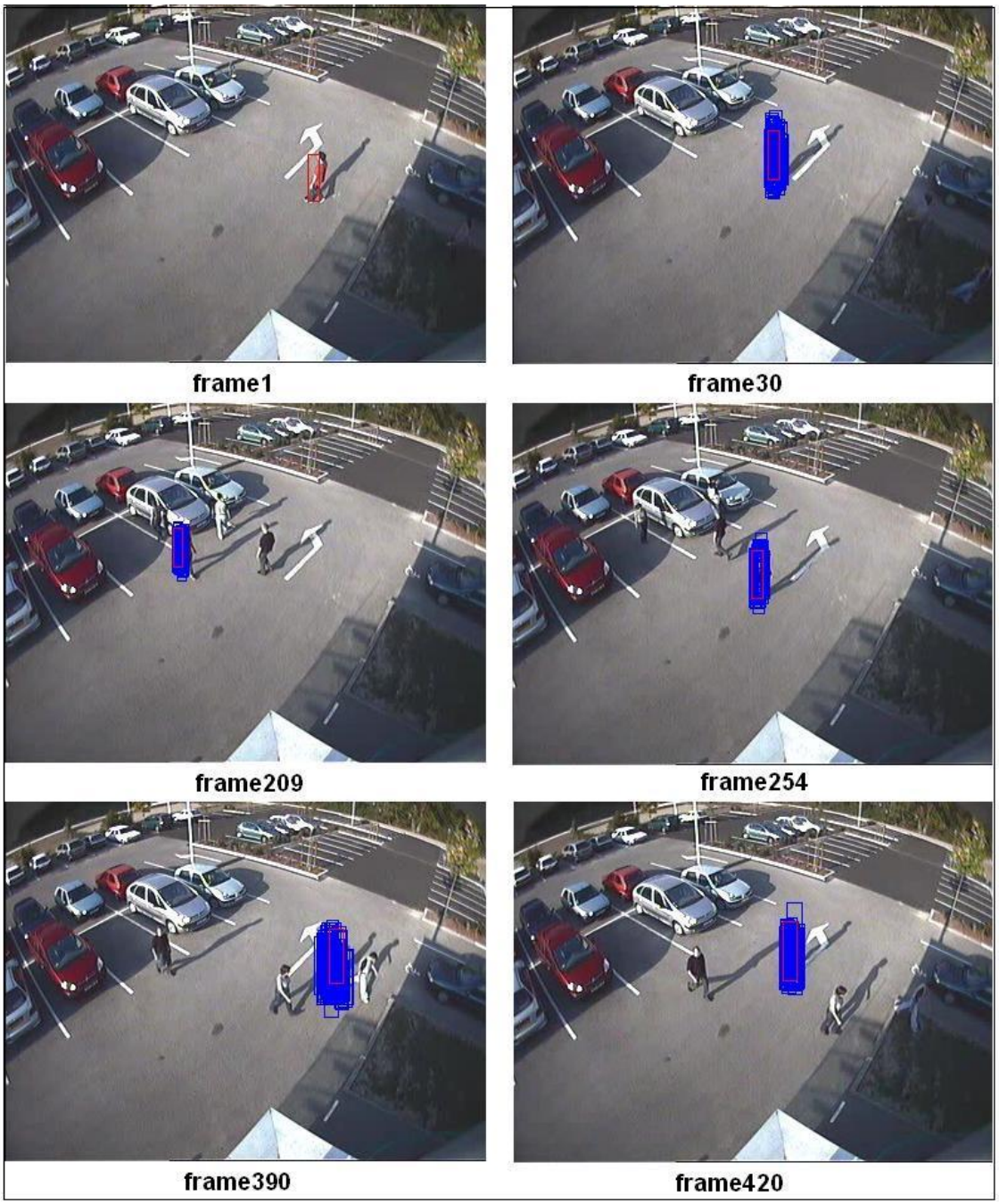


Figure 7.1: Tracking results with displaying all particles

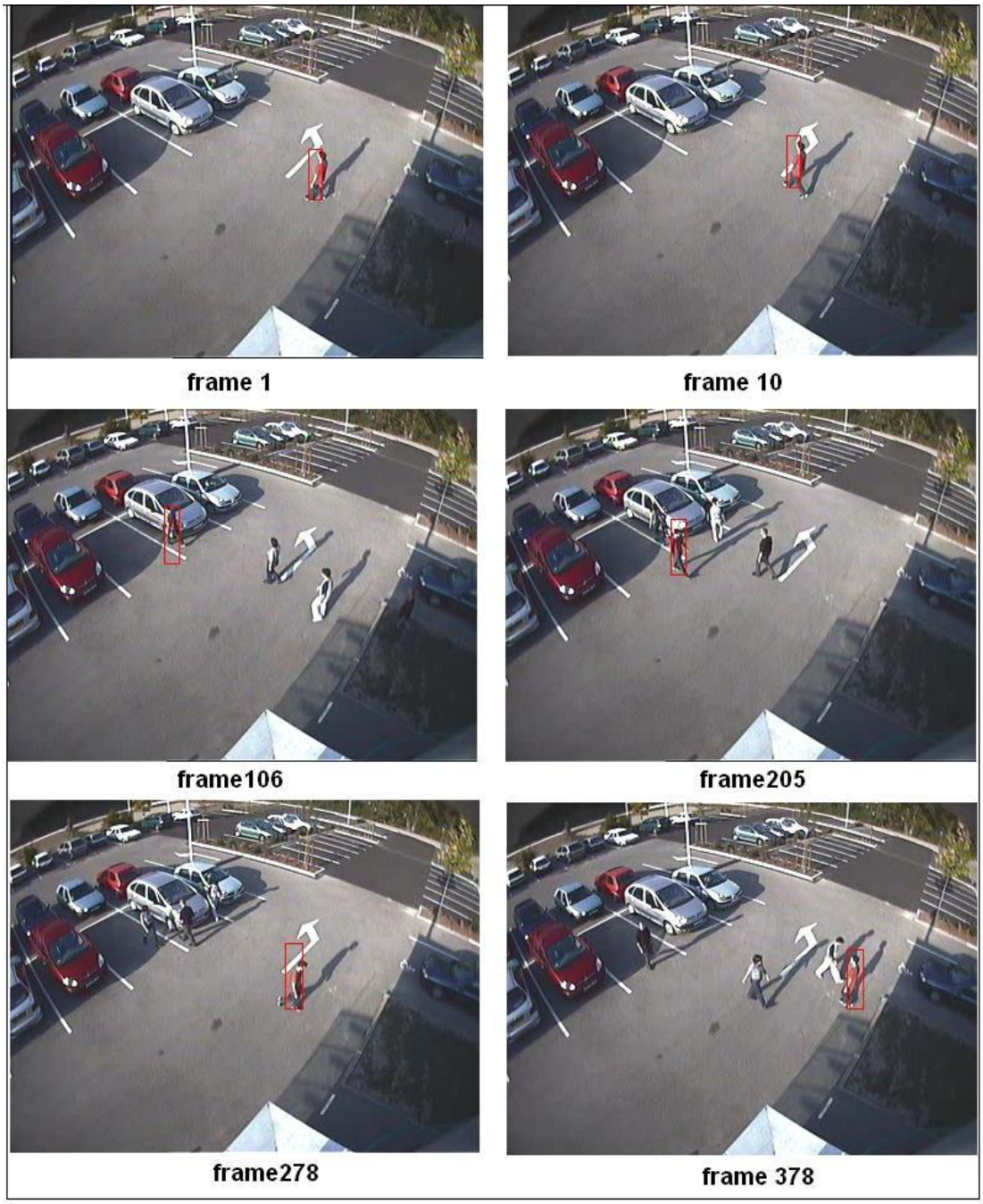


Figure 7.2: Tracking results with displaying the most likely particles

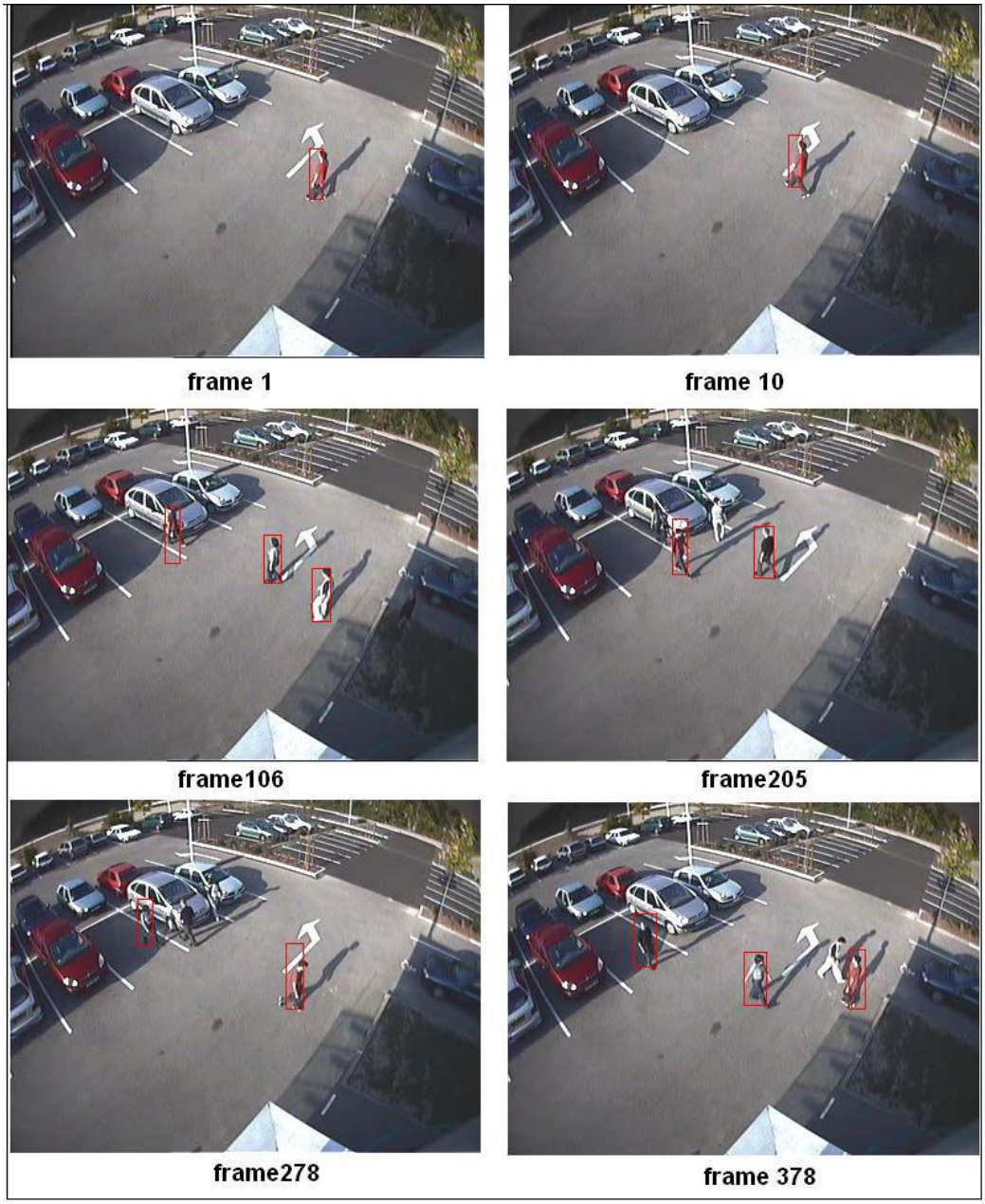


Figure 7.3: Tracker + Adaboost results

Part III

Architecture

Chapter 8

General purpose computation on the GPU

Contents

8.1 Introduction

In this part, we focus on hardware acceleration of computer vision algorithms. As a choice of architecture we selected a low cost solution generally available on all computers. We selected the graphics card. This approach became an attractive solution for computer vision community. To the best of our knowledge, we implemented the first

visual object detection algorithm on the graphics card. The implemented algorithm performs better than the traditional implementation on general purpose processor. It is possible now to implement large detectors and to maintain a real time run.

This part is organized as follow: Chapter 8 presents the graphics hardware as well as the abstraction of the graphics hardware as stream co-processor. Chapter 9 presents the mapping of the visual object detection on the graphics hardware as in [HBC06].

The remainder of this chapter is organized as follows: in Section 8.2 we give an answer to the question *why GPGPU?*. Section 8.3 presents the GPGPU's first generation. The GPGPU's second generation is presented in Section 8.4. Section 8.5 concludes the chapter.

8.2 Why GPGPU?

Commodity computer graphics chips, known generically as Graphics Processing Units or GPUs, are probably todays most powerful computational hardware for the dollar. Researchers and developers have become interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for General-Purpose computing on the GPU). In this chapter we summarize the principal developments to date in the hardware and software behind GPGPU, give an overview of the techniques and computational building blocks used to map general-purpose computation to graphics hardware, and survey the various general-purpose computing tasks to which GPUs have been applied. We begin by reviewing the motivation for and challenges of general-purpose GPU computing. Why GPGPU?

8.2.1 Computational power

Recent graphics architectures provide tremendous memory bandwidth and computational horsepower. For example, the flagship NVIDIA GeForce 7900 GTX ($378 on October 2006) boasts 51.2 GB/sec memory bandwidth; the similarly priced ATI Radeon X1900 XTX can sustain a measured 240 GFLOPS, both measured with GPUBench [BFH04a]. Compare to 8.5 GB/sec and 25.6 GFLOPS theoretical peak for the SSE units of a dual-core 3.7 GHz Intel Pentium Extreme Edition 965 [Int06]. GPUs also use advanced processor technology; for example, the ATI X1900 contains 384 million transistors and is built on a 90-nanometer fabrication process.

Graphics hardware is fast and getting faster quickly. For example, the arithmetic throughput (again measured by GPUBench) of NVIDIAs current-generation launch product, the GeForce 7800 GTX (165 GFLOPS), more than triples that of its predecessor, the GeForce 6800 Ultra (53 GFLOPS). In general, the computational capabilities of GPUs, measured by the traditional metrics of graphics performance, have compounded at an average yearly rate of 1.7 (pixels/second) to 2.3 (vertices/second). This rate of growth significantly outpaces the often-quoted Moore's Law as applied

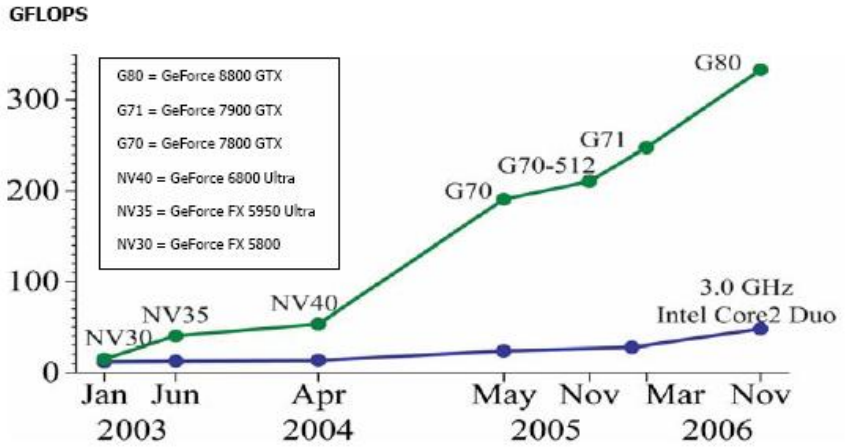


Figure 8.1: CPU to GPU GFLOPS comparison

to traditional microprocessors; compare to a yearly rate of roughly 1.4 for CPU performance [EWN04](Figure 8.1). Why is graphics hardware performance increasing more rapidly than that of CPUs? Semiconductor capability, driven by advances in fabrication technology, increases at the same rate for both platforms. The disparity can be attributed to fundamental architectural differences (Figure 8.2): CPUs are optimized for high performance on sequential code, with many transistors dedicated to extracting instruction-level parallelism with techniques such as branch prediction and out-of-order execution. On the other hand, the highly data-parallel nature of graphics computations enables GPUs to use additional transistors more directly for computation, achieving higher arithmetic intensity with the same transistor count. We discuss the architectural issues of GPU design further in Section 8.3.2.

8.2.2 Data bandwidth

Component	Bandwidth
GPU Memory Interface	35 GB/sec
PCI Express Bus (16)	8 GB/sec
CPU Memory Interface (800 MHz Front-Side Bus)	6.4 GB/sec

Table 8.1: Available Memory Bandwidth in Different Parts of the Computer System

The CPU in a modern computer system (Figure 8.3) communicates with the GPU through a graphics connector such as a PCI Express or AGP slot on the motherboard. Because the graphics connector is responsible for transferring all command, texture, and vertex data from the CPU to the GPU, the bus technology has evolved alongside GPUs over the past few years. The original AGP slot ran at 66 MHz and was 32 bits wide, giving a transfer rate of 264 MB/sec. AGP 2, 4, and 8 followed, each doubling

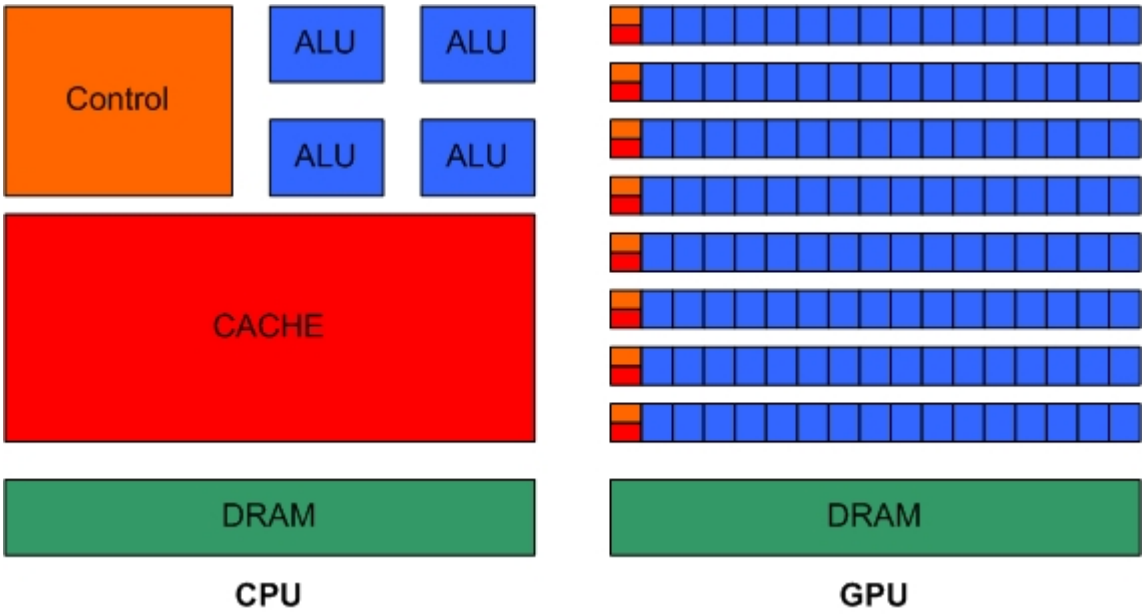


Figure 8.2: CPU to GPU Transistors

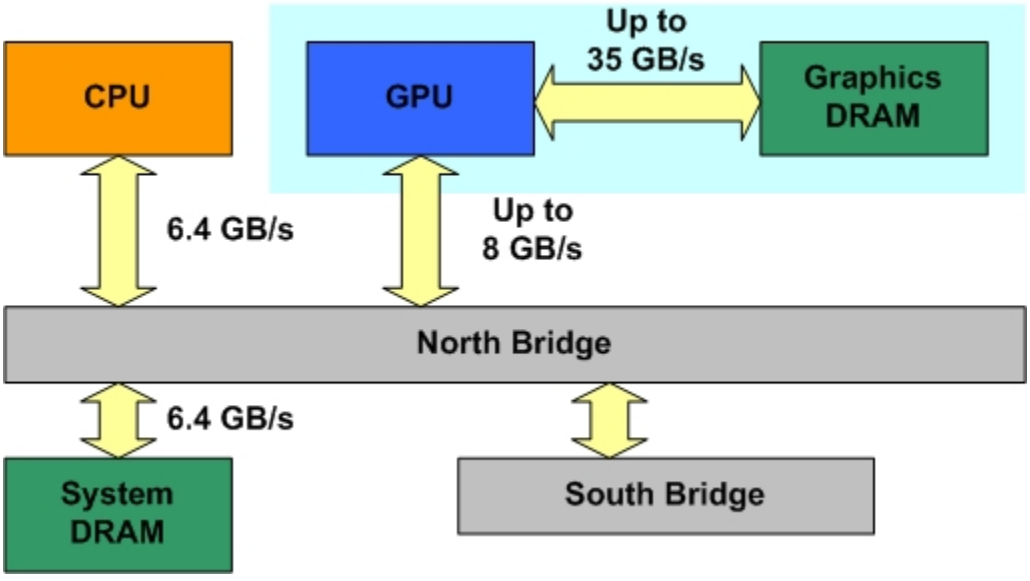


Figure 8.3: The overall system architecture of a PC

the available bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec simultaneously available to and from the GPU. (Your mileage may vary; currently available motherboard chipsets fall somewhat below this limitaround 3.2 GB/sec or less.) It is important to note the vast differences between the GPUs memory interface bandwidth and bandwidth in other parts of the system, as shown in Table 8.1.

Table 8.1 shows that there is a vast amount of bandwidth available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements.

8.2.3 Cost/Performance ratio

Graphics hardware industry is driven by the game market. The price of the top graphics card is about 600$. The motherboard required for the setup of this graphics card is general public and has a low price. Otherwise, the required motherboard for the installation of multiple processors is too high. The graphics card is considered as a free component. This is due to the fact that it is a basic component in a standard PC configuration.

8.3 GPGPU's first generation

8.3.1 Overview

Flexible and precise

Modern graphics architectures have become flexible as well as powerful. Early GPUs were fixed-function pipelines whose output was limited to 8-bit-per-channel color values, whereas modern GPUs now include fully programmable processing units that support vectorized floating-point operations on values stored at full IEEE single precision (but note that the arithmetic operations themselves are not yet perfectly IEEE-compliant). High level languages have emerged to support the new programmability of the vertex and pixel pipelines [BFH.04b,MGAK03,MDP.04]. Additional levels of programmability are emerging with every major generation of GPU (roughly every 18 months). For example, current generation GPUs introduced vertex texture access, full branching support in the vertex pipeline, and limited branching capability in the fragment pipeline. The next generation will expand on these changes and add geometry shaders, or programmable primitive assembly, bringing flexibility to an entirely new stage in the pipeline [Bly06]. The raw speed, increasing precision, and rapidly expanding programmability of GPUs make them an attractive platform for general purpose computation.

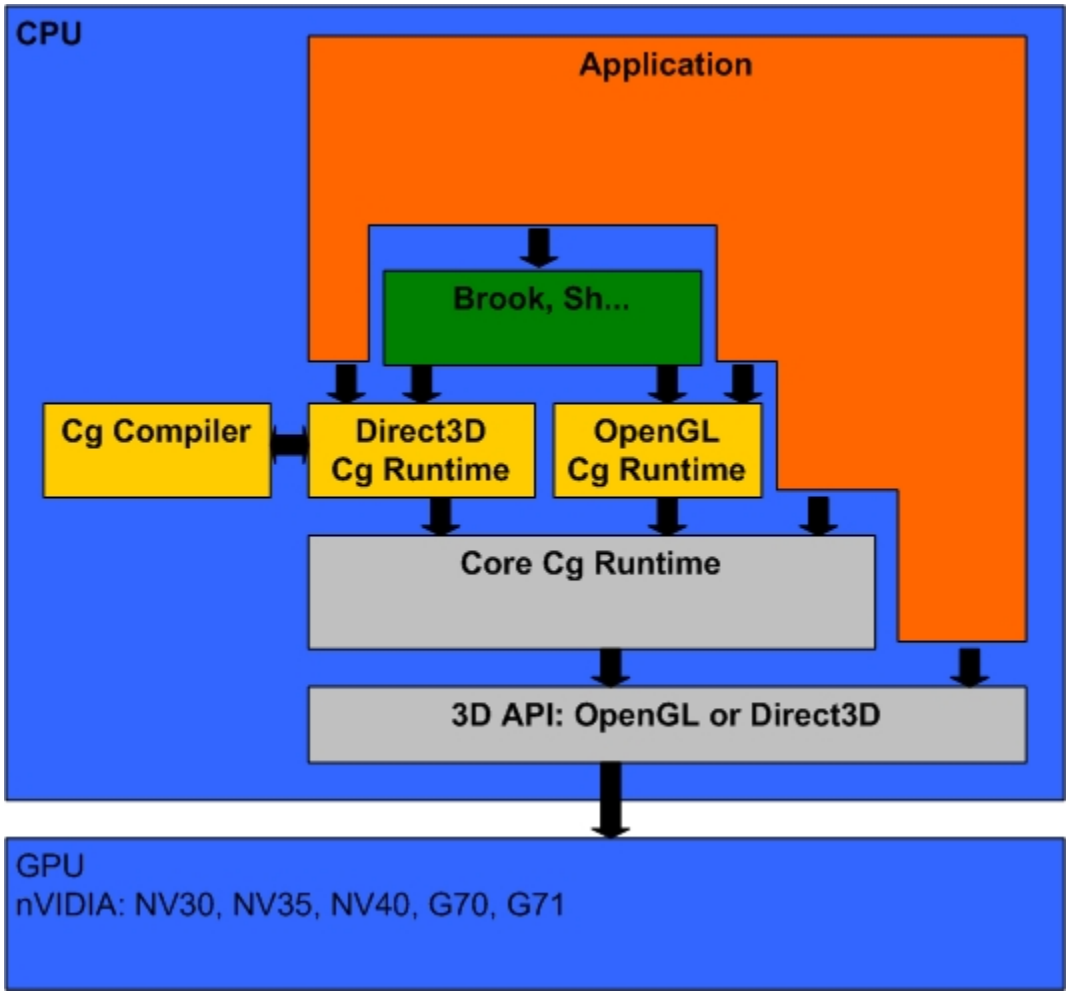


Figure 8.4: First Generation GPGPU framework

Limitations and difficulties

Power results from a highly specialized architecture, evolved and tuned over years to extract maximum performance on the highly parallel tasks of traditional computer graphics. The increasing flexibility of GPUs, coupled with some ingenious uses of that flexibility by GPGPU developers, has enabled many applications outside the original narrow tasks for which GPUs were originally designed, but many applications still exist for which GPUs are not (and likely never will be) well suited.Word processing, for example, is a classic example of a pointer chasing application, dominated by memory communication and difficult to parallelize. Todays GPUs also lack some fundamental computing constructs, such as efficient scatter memory operations (i.e., indexed-write array operations) and integer data operands. The lack of integers and associated operations such as bit-shifts and bitwise logical operations (AND, OR, XOR, NOT) makes GPUs ill-suited for many computationally intense tasks such as cryptography (though upcoming Direct3D 10-class hardware will add integer support and more generalized instructions [Bly06]). Finally, while the recent increase in precision to 32-bit floating point has enabled a host of GPGPU applications, 64-bit double precision arithmetic remains a promise on the horizon. The lack of double precision prevents GPUs from being applicable to many very large-scale computational science problems. Furthermore, graphics hardware remains difficult to apply to non-graphics tasks. The GPU uses an unusual programming model (Section 2.3), so effective GPGPU programming is not simply a matter of learning a new language. Instead, the computation must be recast into graphics terms by a programmer familiar with the design, limitations, and evolution of the underlying hardware. Today, harnessing the power of a GPU for scientific or general-purpose computation often requires a concerted effort by experts in both computer graphics and in the particular computational domain. But despite the programming challenges, the potential benefits leap forward in computing capability, and a growth curve much faster than traditional CPUs are too large to ignore.

The potential of GPGPU

A vibrant community of developers has emerged around GPGPU (http://GPGPU.org/), and much promising early work has appeared in the literature.We survey GPGPU applications, which range from numeric computing operations, to non-traditional computer graphics processes, to physical simulations and game physics, to data mining. We cover these and more applications in Section 5.

8.3.2 Graphics pipeline

The graphics pipeline is a conceptual architecture, showing the basic data flow from a high level programming point of view. It may or may not coincide with the real hardware design, though the two are usually quite similar. The diagram of a typical graphics pipeline is shown in Figure 8.5. The pipeline is composed of multiple functional stages. Again, each functional stage is conceptual, and may map to one or

multiple hardware pipeline stages. The arrows indicate the directions of major data flow. We now describe each functional stage in more detail.

Application

The application usually resides on a CPU rather than GPU. The application handles high level stuff, such as artificial intelligence, physics, animation, numerical computation, and user interaction. The application performs necessary computations for all these activities, and sends necessary command plus data to GPU for rendering.

Host and command

The host is the gate keeper for a GPU. Its main functionality is to receive commands from the outside world, and translate them into internal commands for the rest of the pipeline. The host also deals with error condition (e.g. a new glBegin is issued without first issuing glEnd) and state management (including context switch). These are all very important functionalities, but most programmers probably do not worry about these (unless some performance issues pop up).

Geometry

The main purpose of the geometry stage is to transform and light vertices. It also performs clipping, culling, viewport transformation, and primitive assembly. We now describe these functionalities in detail.

The programmable Vertex Processor

The vertex processor has two major responsibilities: transformation and lighting.

In transformation, the position of a vertex is transformed from the object or world coordinate system into the eye (i.e. camera) coordinate system. Transformation can be concisely expressed as follows:

$$(x_o, y_o, z_o, w_o) = (x_i, y_i, z_i, w_i)M^T$$

Where (x_i, y_i, z_i, w_i) is the input coordinate, and (x_o, y_o, z_o, w_o) is the transformed coordinate. M is the 4×4 transformation matrix. Note that both the input and output coordinates have 4 components. The first three are the familiar x, y, z cartesian coordinates. The fourth component, w, is the homogeneous component, and this 4-component coordinate is termed homogeneous coordinate. Homogeneous coordinates are invented mainly for notational convenience. Without them, the equation above will be more complicated. For ordinary vertices, the w component is simply 1.

Let we give you some more concrete statements of how all these means. Assuming the input vertex has a world coordinate (x_i, y_i, z_i). Our goal is to compute its location in the camera coordinate system, with the camera/eye center located at (x_e, y_e, z_e) in the world space. After some mathematical derivations which are best shown on

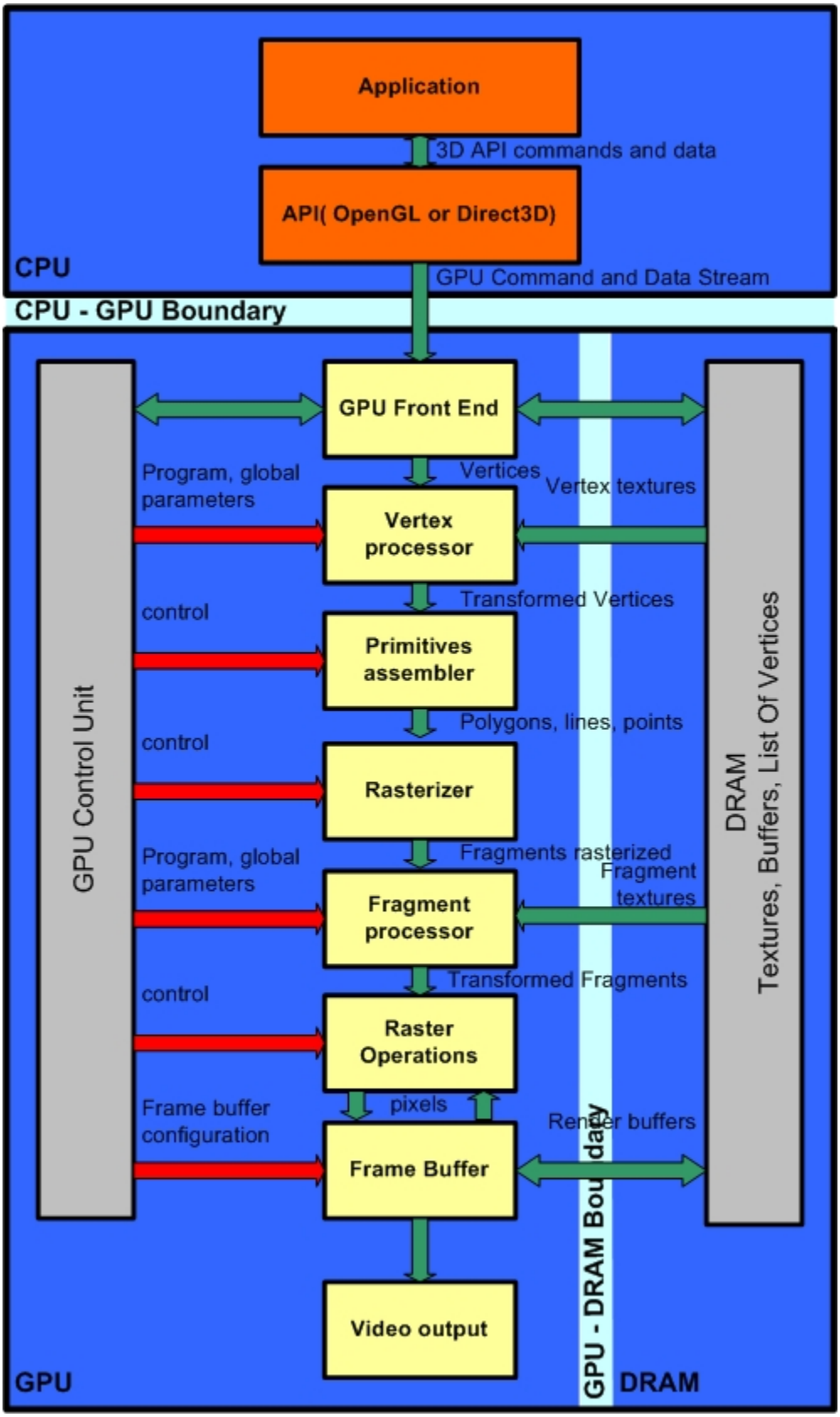


Figure 8.5: Graphics pipeline

a white board, you can see that the 4×4 matrix M above is composed of several components: the upper-left 3×3 portion the rotation sub-matrix, the upper right 3×1 portion the translation vector, and the bottom 1×4 vector the projection part. In lighting, the color of the vertex is computed from the position and intensity of the light sources, the eye position, the vertex normal, and the vertex color. The computation also depends on the shading model used. In the simple Lambertian model, the lighting can be computed as follows:

$$(\overline{n}.\overline{l})c$$

Where $\overline{n}$ is the vertex normal, $\overline{l}$ is the position of the light source relative to the vertex, and c is the vertex color. In old generation graphics machines, these transformation and lighting computations are performed in fixed-function hardware. However, since NV20, the vertex engine has become programmable, allowing you to customize the transformation and lighting computation in assembly code [Lindholm et al. 2001]. In fact, the instruction set does not even dictate what kind of semantic operation needs to be done, so you can actually perform arbitrary computation in a vertex engine. This allows us to utilize the vertex engine to perform non-traditional operations, such as solving numerical equations. Later, we will describe the programming model and applications in more detail.

Primitive assembly

Here, vertices are assembled back into triangles (plus lines or points), in preparation for further operation. A triangle cannot be processed until all the vertices have been transformed as lit. As a result, the coherence of the vertex stream has great impact on the efficiency of the geometry stage. Usually, the geometry stage has a cache for a few recently computed vertices, so if the vertices are coming down in a coherent manner, the efficient will be better. A common way to improve vertex coherency is via triangle stripes.

Clipping and Culling

After transformation, vertices outside the viewing frustum will be clipped away. In addition, triangles with wrong orientation (e.g. back face) will be culled.

Viewport Transformation

The transformed vertices have floating point eye space coordinates in the range $[-1, 1]$. We need to transform this range into window coordinates for rasterization. In viewport stage, the eye space coordinates are scaled and offseted into $[0, height - 1] \times [0, width - 1]$.

Rasterization

The primary function of the rasterization stage is to convert a triangle (or line or point) into a set of covered screen pixels. The rasterization operation can be divided into two major stages. First, it determines which pixels are part of the triangle. Second, rasterization interpolates the vertex attributes, such as color, normal, and texture coordinates, into the covered pixels.

Fragment

Before we introduce the fragment stage, let us first define what fragment is. Basically, a fragment corresponds to a single pixel and includes color, depth, and sometimes texture coordinate values. For an input fragment, all these values are interpolated from vertex attributes in the rasterization stage, as described earlier. The role of the fragment stage is to process each input fragment so that a new color or depth value is computed. In old generation graphics machines, the fragment stage has fixed function and performed mainly texture mapping. Between NV20 and NV30, the fragment stage has become reconfigurable via register combiners. But this is all old stuff and I dont think you need to worry about it. Since NV30, the fragment stage has become fully programmable just like the vertex processor. In fact, the instruction set of vertex and fragment programs are very similar. The major exception is that the vertex processor has more branching capability while the fragment processor has more texturing capability, but this distinction might only be transitory. Given their programmability and computation power, fragment processors have been both embraced by the gaming community for advanced rendering effects, as well scientific computing community for general purpose computation such as numerical simulation [Harris 2005]. Later, we will describe the programming model and applications in more detail.

Raster Operation

ROP (Raster Operation) is the unit that writes fragments into the frame-buffer. The main functionality of ROP is to efficiently write batches of fragments into the frame-buffer via compression. It also performs alpha, depth, and stencil tests to determine if the fragments should be written or discarded. ROP deals with several buffers residing in the frame-buffer, including color, depth, and stencil buffers.

Frame Buffer

The frame-buffer is the main storage for the graphics pipeline, in addition to a few scattered caches and FIFOs throughout the pipe. The frame-buffer stores vertex arrays, textures, and color, depth, stencil buffers. The content of the color buffer is fed to display for viewing. Frame-buffer has several major characteristics: size, width, latency, and clock speed. The size determines how much and how big textures and buffers you can store on chip, the width determines how much data maximum you

can transfer in one clock cycle, the latency determines how long you have to wait for a data to come back, and the clock speed determines how fast you can access the memory. The product of width and clock speed is often termed bandwidth, which is one of the most commonly referred jargon in comparing DRAMs. However, for graphics applications, latency is probably at least as important as bandwidth, since it dedicates the length and size of all internal FIFOs and caches for hiding latency. (Without latency hiding, we would see bubbles in the graphics pipeline, wasting performance.) Although frame-buffer appears to be mundane, it is crucial in determining the performance of a graphics chip. Various tricks have been invented to hide frame-buffer latency. If frame-buffers had zero latency, the graphics architecture would be much simpler; we can have a much smaller shader register file and we don't even need a texture cache anymore.

8.3.3 Programming language

Standard 3D programming interfaces

The graphics hardware is accessible using a standard 3D application programming interface (API). Two APIs are actually available. The first API is DirectX from Microsoft. The second API is OpenGL from Silicon Graphics Incorporated. Both APIs provide an interface to handle GPU. Each API has a different phylosophy. DirectX depends directly on Microsoft's technology. OpenGL is modular and extensible. Each 3D hardware provider can extend OpenGL to support the new functionalities provided by the new GPU. OpenGL is standardized by OpenGL Architectural Review Board (ARB) which is composed of the main graphics constructors.

OpenGL based solutions have a major advantage relative to DirectX based solutions - these solutions are cross platform. This advantage motivated our choice of OpenGL for 3D programming. We were interested in general programming on the GPU. We started our development on Linux platform, and we moved backward to Windows recently.

For more details on these APIs, you can find advanced materials in the literature covering OpenGL and DirectX.

High level shading languages

A shading language is a domain-specific programming language for specifying shading computations in graphics. In a shading language, a program is specified for computing the color of each pixel as a function of light direction, surface position, orientation, and other parameters made available by rendering system. Typically shading languages are focused on color computation, but some shading systems also support limited modeling capabilities, such as support for displacement mapping.

Shading languages developped specifically for programmable GPUs include the OpenGL Shading Language (GLSL), the Stanford Real-Time Shading Language (RTSL), Microsoft's High-level Shading Language (HLSL), and NVIDIA's Cg. Of these lan-

guages, HLSL is DirectX-specific, GLSL is OpenGL-specific, Cg is multiplatform and API neutral, but developed by NVIDIA (and not well supported by ATI), and RTSL is no longer under active development.

8.3.4 Streaming model of computation

Streaming architecture

Stream architectures are a topic of great interest in computer architecture. For example, the Imagine stream processor demonstrated the effectiveness of streaming for a wide range of media applications, including graphics and imaging. The Stream/KernelC programming environment provides an abstraction which allows programmers to map applications to the Imagine processor.

Although, these architectures are specific and cope well with computer vision algorithms previously presented in Part II. These architectures will not be our principal interest. Otherwise, we mention these architectures to give a total sight of the problem. Since, as we described our motivation in Part I, we are interested particularly in architectures with similar functionalities which are available for general public graphics hardware.

Stream programming model

The stream programming model exposes the locality and concurrency in media processing applications. In this model, applications are expressed as a sequence of computation kernels that operate on streams of data. A kernel is a small program that is repeated for each successive element in its input streams to produce output streams that are fed to subsequent kernels. Each data stream is a variable length collection of records, where each record is a logical grouping of media data. For example, a record could represent a triangle vertex in a polygon rendering application or a pixel in an image processing application. A data stream would then be a sequence of hundreds of these vertices or pixels. In the stream programming model, locality and concurrency are exposed both within kernel and between kernels.

Definition 13 (Streams)**.** *A stream is a collection of data which can be operated on in parallel. A stream is made up of elements. Access to stream elements is restricted to kernels (Definition 14) and the streamRead and streamWrite operators, that transfer data between memory and streams.*

Definition 14 (Kernels)**.** *A kernel is a special function which operates on streams. Calling a kernel on a stream performs an implicit loop over the elements of the stream, invoking the body of the kernel for each element.*

Definition 15 (Reductions)**.** *A reduction is a special kernel which accepts a single input stream. It provides a data-parallel method for calculating either a smaller stream of the same type, or a single element value.*

8.3.5 Programmable graphics processor abstractions

From abstract point of view, the GPU is a streaming processor (Figure 8.6), particularly suitable for the fast processing of large arrays [SDK05]. Thus, graphics processors have been used by researchers to enhance the performance of specific, non-graphic applications and simulations. Researchers started to use the graphics hardware for general purpose computation using traditional graphics programming languages: a graphics Application Programming Interface (API), and a graphics language for the kernels well known as Shaders. In Appendix A we describe a hello world example of using the GPU for general purpose computing, using graphics APIs and Shading language.

The graphics API was previously described in Section 8.3.3. It understands function calls used to configure the graphics hardware. Implementations of these APIs are available for a variety of languages, e.g. C, C++, Java, Python. These APIs target always graphics programming and thus one must encapsulate the native API in a library to make the code looks modular.

As described in Section 8.3.3, shading languages can be used with DirectX (HLSL) or OpenGL (GLSL), or both of them (Cg). The programming model of a Shader is described in Section 8.3.5.

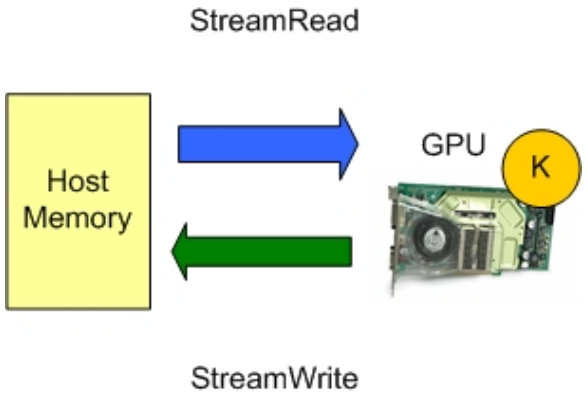


Figure 8.6: The graphics hardware abstraction as a streaming coprocessor

It is not always easy to use graphics APIs and Shading languages for non graphics community. That is why several projects try to further abstract the graphics hardware as a slave coprocessor, and to develop a suitable model of computation. Ian Buck et al. [BFH+04b] describe a stream programming language called Brook for GPU. Brook extends C++ to provide simple data-parallel constructs to allow using the GPU as a streaming coprocessor. It consists in two components: a kernel compiler *brcc*, which compiles kernel functions into legal Cg code, and a *runtime system* built on top of OpenGL (and DirectX) which implements the Brook API. In Appendix B we provide a hello world example of Brook for image processing similar to Appendix A.

The programming model of fragment processor

The execution environment of a fragment (or vertex) processor is illustrated in Figure 8.7. For every vertex or fragment to be processed, the shader program receives from the previous stage the graphics primitives in the read-only input registers. The shader is then executed and the result of rendering is written on the output registers. During execution, the shader can read a number of constant values set by the host processor, read from texture memory (latest GPUs started to add the support for vertex processors to access texture memory), and read and write a number of temporary registers.

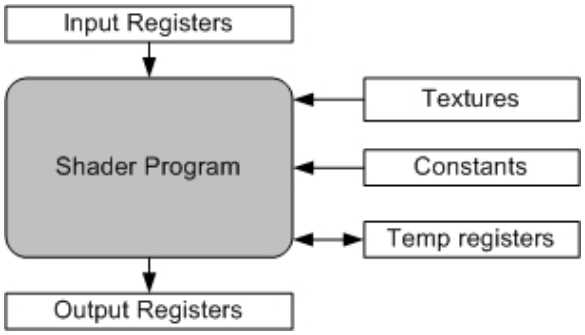


Figure 8.7: Programming model for current programmable graphics hardware. A shader program operates on a single input element (vertex or fragment) stored in the input registers and writes the execution result into the output registers.

Implementation constraints

This section discusses basic properties of GPUs which are decisive for the design of efficient algorithms on this architecture. Table 8.2 presents a summary of these properties. Note that these properties depend on the graphics hardware and are changing each 6 months. In Table 8.2 we consider the nVIDIA G7 family and the first Geforce FX (5200) [nVI07a].

Property	CPU	GFX5200	GF7800GTX
Input Streams	native 1D	native 1D, 2D, 3D	native 1D, 2D, 3D
Output Streams	native 1D	native 2D	native 2D
Gathers	arbitrary	arbitrary	arbitrary
Scatters	arbitrary	global	global, emulated
Dynamic Branching	yes	no	yes
Maximum Kernel Length	unlimited	1024	unlimited

Table 8.2: Graphics hardware constraints

In the new GPUs generation these properties completely changed. This global change is made by the fact that the GPU emerged recently to a general multiprocessor

on chip architecture. This will be highlighted in the next section.

8.4 GPGPU's second generation

Figure 8.8 shows a new GPGPU diagram template. This new diagram is characterized by a novel approach: the GPGPU application bypasses the graphics API (OpenGL, DirectX). This novel functionality is provided by CUDA. CUDA is the new product released by nVIDIA. CUDA stands for **Compute Unified Device Architecture** and is a new hardware and software architecture issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API.

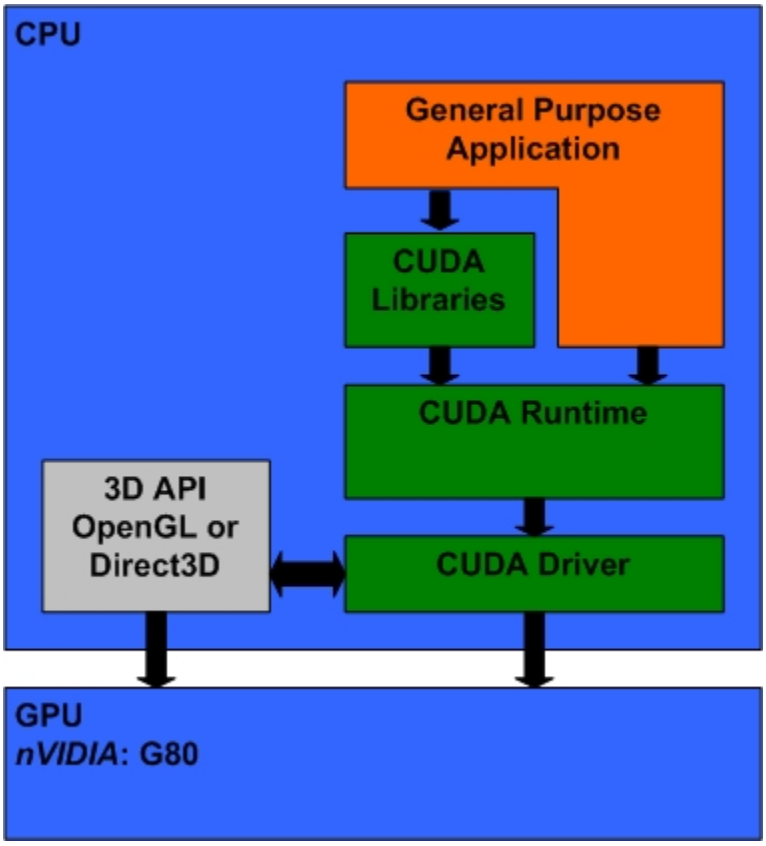


Figure 8.8: Second Generation GPGPU framework

The CUDA software stack is illustrated in Figure 8.8. It consists in several layers: a hardware driver, an application programing interface (API) and its runtile, and two high-level mathematical libraries of common usage, CUFFT and CUBLAS.

The CUDA API is an extension of the ANSI C programming language for a minimum learning curve.

The operating system's multitasking mechanism is responsible for managing the access to the GPU by several CUDA and graphics applications running concurrently.

CUDA provides general DRAM memory addressing and supports an arbitrary scatter operation.

The CUDA software will have a great impact on the GPGPU research. Researchers can now concentrate on writing applications on a multiprocessor using standard C programming language. They don't have to try to abstract the graphics hardware through graphics APIs and shading languages. Researchers can now put their effort on mapping their algorithms on the data-parallel architecture using a data-parallel model of computation. This justifies the decomposition of the GPGPU into two generations.

In the remainder sections we will give you an overview of the CUDA software based on CUDA documentation [nVI07b]. The reader is invited to read the documentation for more details on the architecture. CUDA was released recently, and we did not use it in our work. We mention it in this thesis with the aim to show the importance as well as the evolution of the GPGPU approach. GPGPU is now supported by the graphics hardware constructor nVIDIA itself.

8.4.1 Programming model

CUDA models the GPU as a computing device capable of executing a very high number of threads in parallel. The GPU operates as a coprocessor to the host CPU. Both, the host and the device maintain their own DRAM, referred to as host memory and device memory respectively. DMA engines are used to accelerate the memory copy transactions between the two DRAMs.

The device is implemented as a set of multiprocessors on chip. Each multiprocessor consists in several SIMD processors with on-ship shared memory. At any given clock cycle each processor executes the same instruction, but operates on different data.

The adequate programming model for this architecture is as follows: the thread is the elementary unit in the execution model. Threads are grouped into batch of threads called blocks. Blocks are grouped together and form what is called grids. Thus, each thread is identified by its grid id, its block id and its id within the block.

Only threads in the same block can safely communicate. The maximum number of threads in a block, and the maximum number of blocks in a grid, as well as the number of grids in the device are parameters of the implementations. These parameters are varying from a model to another.

8.4.2 Application programming interface (API)

The goal of the CUDA programming interface is to provide for C programmers the possibility to easily write a program for execution on the device. The implementation of algorithms on the device with CUDA is out of the focus of the present work, otherwise, we provide in Appendix C a hello word example of a CUDA program for image processing similar to the previous hello worlds for Brook and OpenGL/Cg. For more information on the application programming interface, the reader is invited to refer to CUDA programming guide [nVI07b] (Chapter 4).

8.5 Conclusion

In this chapter we introduced the architecture of the graphics hardware. This architecture is in a migration phase: from an architecture completely based on the graphics pipeline toward a generic architecture based on a programmable multiprocessor on chip. This migration is motivated by the recent usage of the graphics hardware for general purpose computing.

In this thesis, we experimented with the first generation of the programmable graphics hardware (GeforceFX5200, GeforceFX6600GT and GeforceFX7800GTX). Thus we had to work with graphics APIs (OpenGL) and shaders (Cg) to implement programs for general purpose computing. We introduced research activities launched for the best abstraction of the graphics hardware as a streaming coprocessor (Brook).

Chapter 9

Mapping algorithms to GPU

Contents

9.1 Introduction

In this chapter we describe our experience on writing computer vision applications (see Part II) for execution on GPU. We started the GPGPU activity since 2004 from scratch. What do we mean by scratch, that we were newbies for GPU programming without any previous experience with shading languages and a little experience with OpenGL. We were motivated by the performance of these architectures and with possibility to implement parallel algorithms on a commodity PC.

Our first source of inspiration was OpenVidia [FM05] where Fung and Mann presented an innovative platform which uses multiple graphics cards to accelerate computer vision algorithms. The platform was targeted to a Linux platform only. We took this platform in hands and we succeeded to make it operational after several weeks of configuration on a Fedora 4 box, with GeforceFX5200 on AGP bus and three GeforceFX5200 on PCI bus. We used OpenGL and Cg during this part of work. This first part was for us a training period for graphics APIs as well as for Cg Shading language. The graphics cards market was and still is too dynamic, and we found that new graphics cards (Geforce5900, Geforce6800...) performs better than

the whole OpenVIDIA platform. And no more graphics cards provided for the PCI bus. Thus we left back the OpenVIDIA platform and we focused on single graphics card architecture.

Once we became familiar with graphics languages, we started to search for more abstraction of the GPUs. Our objective was to find an abstraction of the GPU with the aim to implement portable applications for several graphics cards models. And to design applications for a stream model of computation. This will be useful for the future implementation of these applications for execution on stream processors. We found several approaches (see Section 8.3.5) and we were motivated for Brook [BFH+04b]. Brook initially developed for stream processors and adapted for GPUs by Buck et al. was a good choice because it copes with our objectives: Brook applications are cross platform, and could be compiled to run on several architectures (nv30, nv35, nv40). Brook applications are stream based and give a stream coprocessor abstraction of the GPU.

We implemented image processing filters available in Camellia image processing library [BS03]. These filters perform better on GPU than the CPU. The main handicap was the communication time on the AGP bus. For this first part we had not a special complete application to accelerate. The idea was to accelerate the Camellia library on the GPU. And thus any application based on Camellia could benefit from the GPU acceleration. We accelerated mathematical morphology operators, linear filters, edge detection filters and color transformation filters. This work is comparable to another approach elaborated by Farrugia and Horain [FH06]. In their work Farrugia and Horain tried to develop a framework for the execution of OpenCV filters on the GPU using OpenGL and GLSL shading language. In their release, they provided the acceleration of the previous described filters, and they developed a framework comparable to OpenVIDIA [FM05]. This approach requires graphics programming knowledge to add a new filter. It is too dependent on the subsequent architecture, and can't resolve automatically hardware constraints: register number, shader length, output registers... Thus we find that the usage of Brook to accelerate a stack of algorithmic filters could be a more durable solution and could be adapted to new technologies bringed up by the new GPUs.

After image processing filters we focused in more complex algorithms which can't perform in real time on standard PC. We were particularly interested in visual object detection with sliding window technique. This algorithm requires some optimizations to perform in real time, and these optimizations decrease the final accuracy as described in Abramson's thesis [Abr06].

We developed a framework for mapping the visual detectors on the GPU. To the best of our knowledge we were the first to implement a sliding window detector for visual objects on the GPU [HBC06]. The first framework was based on OpenGL/Cg and targeted to nv30 based GPU (Geforce5200). This framework performs poorly on the Geforce5200, but it performs in real time on newer nv40 based GPUs as Geforce6600. We also developed an implementation of the framework using Brook. The Brook framework has similar performance to the previous one.

In this chapter we focus on mapping visual object detectors on the GPU. The remainder of this chapter is organized as follows: Section 9.2 presents the design of the visual detector as a stream application. Section 9.3 presents the hardware constraints and their impact on the stream application design. Section 9.4 presents the code generator framework. A performance analysis task is described in Section 9.5. Section 9.6 concludes the chapter.

9.2 Mapping visual object detection to GPU

In this section we focus on the design of the algorithm using a stream model of computation previously described in Section 8.3.4.

Some parts of the application are not suitable for a streaming model of computation. Thus the first question while writing the application is: which computation to move to the GPU?. To answer to this question we need to study the code of the application searching for possible functions that could be accelerated by the GPU and more generally the computation part which is rich in data parallelism.

This is a pure algorithmic activity which consists in modeling the application using streams and kernels. We adopted a hierarchical approach. This approach consists in identifying the initial streams, and the main functional parts as blocks with inputs and outputs. These inputs and outputs represent data streams.

We consider the visual detector as described in Section 6.3.6. We consider the global processing pattern (see Figure 6.17(a)). For this first implementation we consider the Control-Points features as visual weak classifiers. We made this choice because we used this configuration for a large set of problems, and it was under our focus. Therefore, the major part of the subsequent analysis can be used for other type of features, and only the preprocessing part will change.

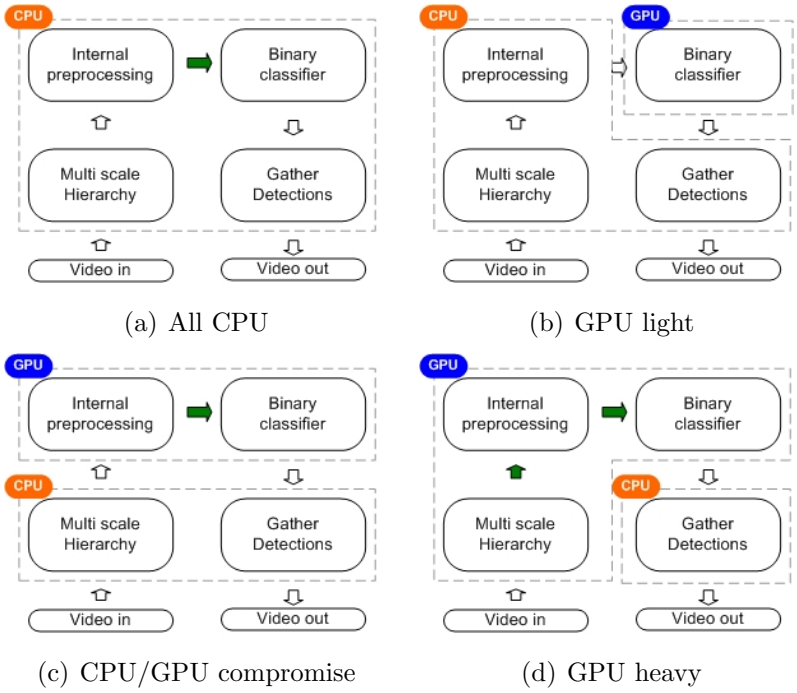
While reviewing the main parts of the detector based on the Control-Points features (Figure 9.1) we distinguish the following operations:

Internal Resolution Tree The Control-Points Features are applied to the three resolutions of the input frame. This operation is undertaken for each input frame.

Preprocessing The preprocessing step consists in preparing the three resolutions of the input image as described in Section 6.3.3.

Binary Classifier The binary classifier is a pixel wise operation. It consists in a sliding window operation, at each pixel of the input frame, we test the possibility to have an object at this position defined by its upper left corner and object dimensions.

Detection Grouping Due to multiple detections for a given object at a given position, the gather operation is applied to group multiple detections corresponding to an unique object.



(a) All CPU (b) GPU light

(c) CPU/GPU compromise (d) GPU heavy

Figure 9.1: Application codesign, split between CPU and GPU

The adequation between the CPU/GPU architecture and the visual detector algorithm can be represented by the four logical partitions as follows:

All CPU (see Figure 9.1(a)) All the functional blocks are executed on the CPU. This is the case of the golden reference of the detector.

Light GPU (see Figure 9.1(b)) It consists in executing the classification operation on the GPU. The preprocessing and the post processing are done on the CPU. The hierarchical tree is constructed on the host processor. This is the minimal configuration for the GPU.

CPU/GPU Compromise (see Figure 9.1(c)) If the preprocessing copes well with the data parallel architecture then the preprocessing is mapped to the GPU and then we can obtain a compromised functional decomposition between CPU and GPU.

Heavy GPU (see Figure 9.1(d)) The graphics hardware provides functionalities for building multiscale image representations, so only the post processing operation is executed on the CPU.

The internal resolution tree could be accelerated using a traditional API from OpenGL for rendering a texture to a given Quad of Size smaller than the dimensions of the given texture. This can be done using specific GL Options.

The gathering operation is a global operation and not adapted to data parallelism. It will be implemented on the CPU side.

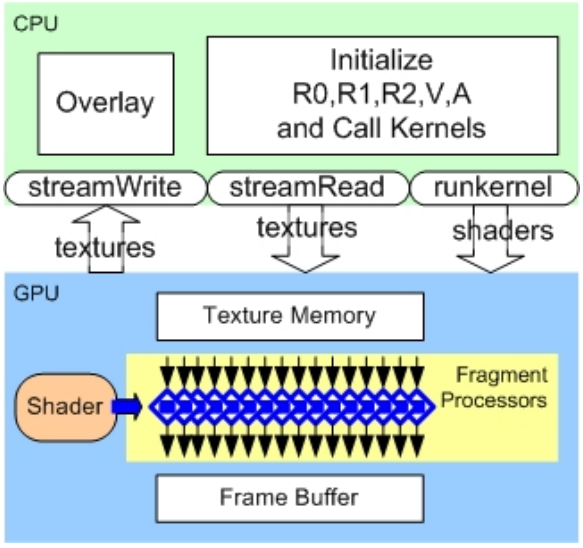


Figure 9.2: Graphics Pipeline used as Streaming Coprocessor. Three services are available: StreamRead, StreamWrite and RunKernel

We adopted the light GPU (see Figure 9.1(b)) partition. The main part moved to the GPU is the binary classifier which consists in applying the same computation to each pixel in the input frame. We present in Algorithm 5 the implementation of the cascade as a streaming application. The input streams are R_0,R_1,R_2 and V. The different layers are called successively and the result of each layer is transmitted to the next layer using the V stream. This part of the implementation is running on the CPU (Figure 9.2). In Algorithm 6 we present the pseudo-code of the implementation of the layer into several kernels (shaders) running on the GPU (Figure 9.2).

Algorithm 5 Cascade of classifiers

Require: Intensity Image R_0
Ensure: a voting matrix

1: Build R_1 and R_2
2: Initialize V from Mask
3: StreamRead R_0, R_1, R_2 and V
4: **for all** i such that $0 \leq i \leq 11$ **do**
5: $\quad V \Leftarrow \text{RunLayer } i, R_0, R_1, R_2, V$
6: **end for**
7: StreamWrite V

9.3 Hardware constraints

The Graphics Hardware has many constraints on the shaders. On the graphics cards implementing NV30 shaders, shader size is limited to 1024 instructions and the con-

Algorithm 6 Layer: Weighted Sum Classifier

Require: Iterator, R_0, R_1, R_2 and V
Ensure: a voting stream.

```
1: if V[Iterator] = true then
2:    S ⇐ 0
3:    for each feature F_j in the layer do
4:       A ⇐ RunFeature j, R_0, R_1, R_2, V
5:       S ⇐ S + A∗ WeightOf(F_j)
6:    end for
7:    if S ≤Threshold then
8:       return false
9:    else
10:      return true
11:   end if
12: else
13:   return false
14: end if
```

stant registers are limited up to 16. On more advanced graphics cards implementing NV40, shader size is unlimited, but the constant registers are limited to 32. These constraints have an influence on the design of the application in terms of streaming application. Thus, to implement a cascade of binary classifiers, we were obliged to decompose the cascade into several kernels, each kernel corresponds to a layer. To achieve an homogenous decomposition of the application, we decomposed each layer into several kernels, called scans, and the whole cascade is the equivalent to a list of successive scans as shown in Figure 9.3. As shown in Figure 9.3, the data transmission between layers is done using the V Stream, and the intra-layer data transmission is done using the A stream to accumulate the intermediate features calculations.

9.4 Code generator

Our code generator is implemented in the Perl script language to generate BrookGPU and Cg/openGL programs according to input Adaboost learning knowledge.

Brook Implementation

We decided to generate the code of the application in BrookGPU language, mainly for two reasons. First, the generated code should be portable to various architectures, even future architectures that are not yet defined. Generating high level programs will allow fundamental changes in hardware and graphics API as long as the compiler and runtime for high level language compilers keep up with those changes. Second, the transparency provided by Brook, which makes it easier to write a streaming

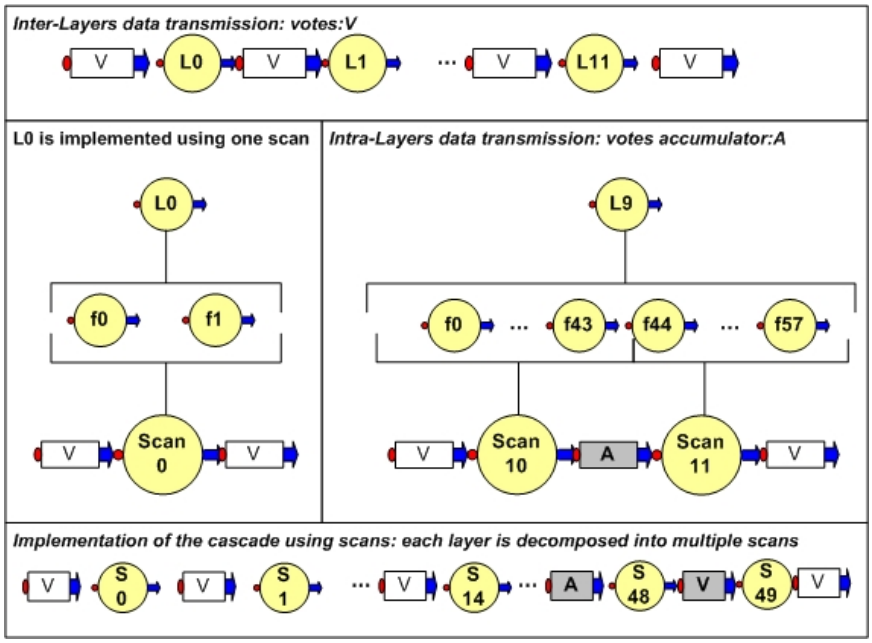


Figure 9.3: Multiscans technique. We consider the CSFD detector. Layer L_0 is composed of two features, it is implemented using only one scan S_0. Layer L_9 is composed of 58 features, it is implemented using two scans S_{10} and S_{11}. S_{10} produces an intermediate voting result, and S_{11} produces the vote of L_9

application. The programmer can concentrate on the application design in terms of kernels and streams without any focus on the Graphics related languages and libraries (Cg, openGL). In Figure 9.4 we show the final system developed with BrookGPU. The Adaboost knowledge description serves as an input to the Code Generator which generates the Brook kernels as well as the main C++ code for streams initialization and read/run/write calls handling. The generated code is a ".br", this file is compiled by brcc to generate the C++ code as well as the assembly language targeted to the GPU. This C++ file is then compiled and linked to the other libraries to build the executable. The main disadvantage of the brook solution is that we are obliged to enter the compile/link chain each time we modify the Adaboost knowledge input.

Cg/OpenGL Implementation

In order to test the performance of the BrookGPU runtime, as well as to benefit from additional features provided by the OpenGL library, we decided to implement a hand written version of the application using Cg to code the kernels as pixel shaders, and we used OpenGL as an API to communicate with the graphics card. The advantage of this solution is that we are not obliged to enter the compile/link chain once we change the input Adaboost knowledge. In fact, the Cg code generator generates the Cg code used to represent the visual detector, and this code is loaded dynamically using the Cg library. The Cg library enables loading Cg code on runtime from files

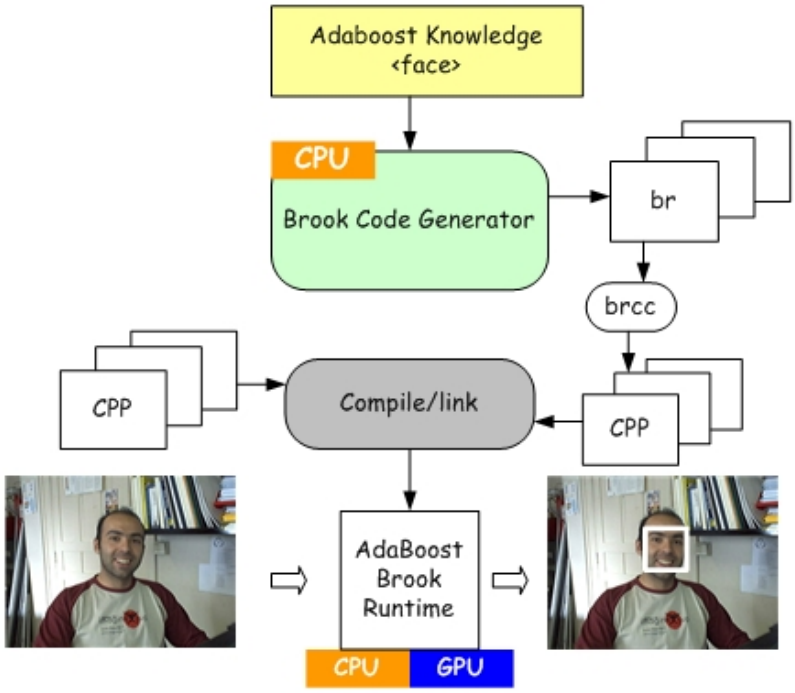


Figure 9.4: Final System, Brook Runtime

stored on disk. The abstraction of the graphics hardware provided by Brook, is not present, so we have to manage our data using textures and our kernel using pixel shaders manually.

9.5 Performance analysis

We tested our GPU implementation on a nVIDIA Geforce 6600GT on PCI-Express. The host is an Athlon 64 3500+, 2.21 Ghz with 512M DDR.

9.5.1 Cascade Stages Face Detector (CSFD)

As shown in Figure 9.6, the CPU version of the CSFD spends most of its time on the first 6 layers, which is related to the fact that the number of windows to test at these layers is still considerable. In the last 6 layers, even if the layers are too complex, the number of windows to classify is not great. Conversely, the GPU implementation spends most of its time on the last layers, and less time on the first 6 layers. This is related to the fact that, in the first layers, the number of features is not too big, so each layer requires a single pass on the GPU, which is too fast. But, the last layers are too complex, and require up to 30 scans per layer (layer 11), so the GPU spends more time classifying the frame.

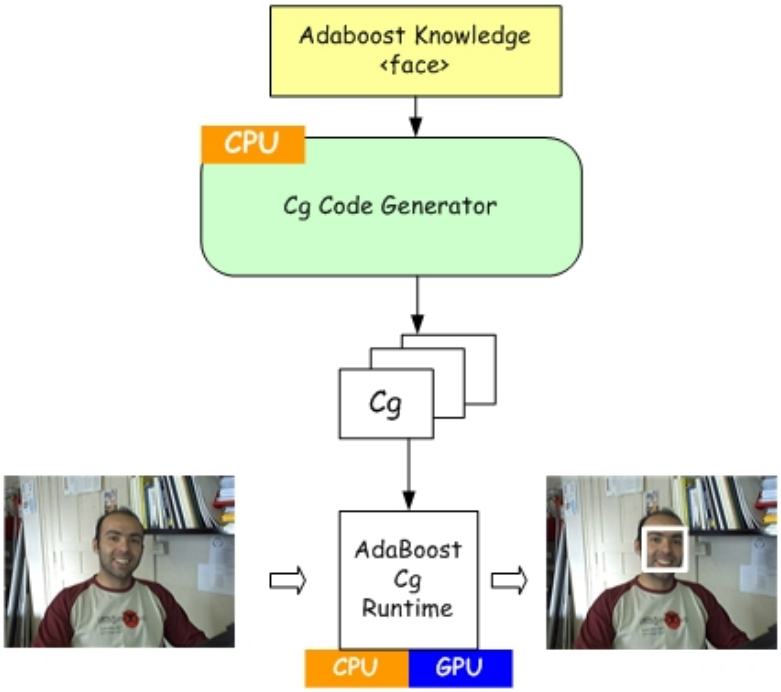


Figure 9.5: Final System, Cg/OpenGL Runtime

In Figure 9.7(a), we present a speedup for each layer of the cascade. We deduce that for the first 6 layers, the GPU is running faster than on CPU; this is due to the high data parallelism support on the GPU. The last 6 layers are running faster on the CPU, due to the small number of windows to classify and the high number of features within the layers.

In Figure 9.7(b) we present the profiling of the mixed solution: the first 6 layers are running on the GPU and the next 6 layers are running on the CPU. Using this decomposition, we reach a real time classification with 15 fps for 415x255 frames.

9.5.2 Single Layer Face Detector (SLFD)

As shown in Table 9.1, on the x86 processor, the SLFD with 1600 features requires 18.8s to classify 92k windows in a 415x255 frame. The CSFD with 12 layer, needs only 175ms (Table 9.1).

The implementation of the SLFD on the GPU requires 2s to classify 100k windows in a 415x255 frame. Which means that the GPU produces a speedup of 9.4 compared to the pure CPU implementation, but it is still not running in real time.

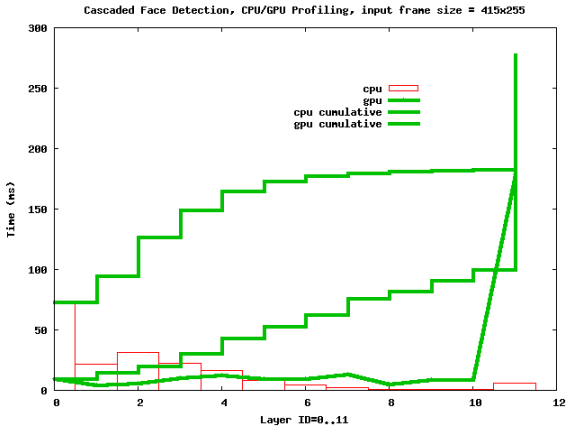
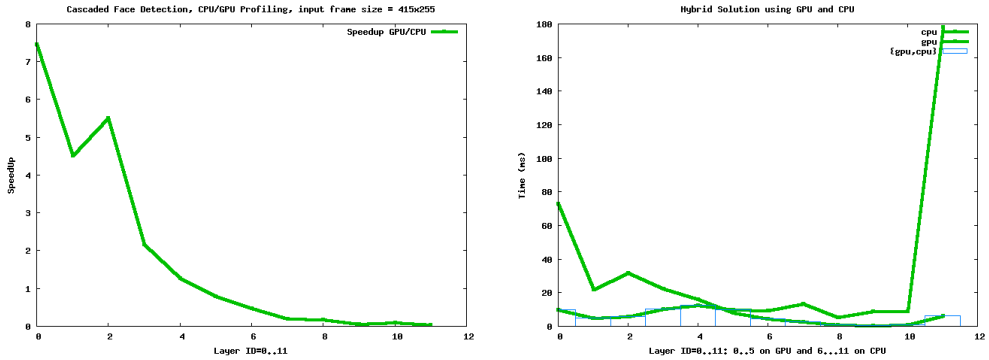


Figure 9.6: Processing Time Profiling



(a) (b)

Figure 9.7: (a) Cascaded Face Detection, Speedup plot for GPU to CPU implementation. We show that the GPU performs better than CPU for the first five layers; (b) Processing Time Profiling for the heterogeneous solution

9.6 Conclusion

In this chapter we described the mapping of the visual object detector to the graphics hardware. We designed the algorithm using the streaming model of computation. We identified the streams and the kernels. Then, we explained how we generated the code for the kernels and how we handled the storage of the images in streams. We faced some technical challenges. We explained these challenges, specially with the limited graphics hardware (GeforceFX5200): memory size and shader length.

We accomplished a successful mapping of the algorithm. We considered a complex knowledge result issued from LibAdaBoost learning task. This knowledge is for face detection based on the control point features. The methodology that we have used in the mapping is generic enough to support other kind of detectors with other type of features.

In the present chapter, we demonstrated that, for a given visual detector, we can

	Time(ms)	ClassificationRate(WPS)
CPU	18805.04	4902.73
GPU	2030.12	53122.06

Table 9.1: Single Layer FD with 1600 features, the GPU implementation is 10 times faster than the CPU implementation

associate several runtimes: the CPU runtime and the GPU runtime. The performance tests showed that the GPU architecture performs better than the traditional CPU for the visual detector.

We can conclude that the GPU architecture with a streaming model of computation modeling is a promising solution for accelerating computer vision algorithms.

Part IV

Application

Chapter 10

Application: PUVAME

Contents

10.1 Introduction

In this part we focus on applications. An application is the adequation of a given algorithmic pipeline and a given architecture. The general framework for intelligent video analysis described in Chapter 4 will be parametrized to cope with the requirements. Some of the optimizations presented in Chapter 8 could occur if the architecture is composed of programmable graphics cards.

In this chapter we present an application for intelligent video analysis which consists in people detection in urban area. It consists in the analysis of video streams from fixed cameras. This analysis aims at detecting people in the video and communicating this information to a global fusion system, which performs tracking and collision estimation.

This application is an integrated part of the French PREDIT[1] project PUVAME[2]. A non exhaustive list of our contributions to this project is as follows:

- We contributed to the different technical meetings around requirements definition.
- We contributed to the design of the application using RTMaps[3] as a development and deployment environment.
- We provided the people detection module based on adaptive background subtraction and machine learning for people detection.
- We created datasets for synchronized multiple cameras acquisition.

The remainder of this chapter is organized as follows: in next section, we give a brief overview of the PUVAME project. In Section 10.3 we detail the accident analysis done by Connex-Eurolum in 2003 and also we describe the chosen use cases. Section 10.4 presents the experimental platform used to evaluate the solutions we propose. Section 10.5 details the architecture of the system. Experimental results are reported in section 10.6. We give some conclusions and perspectives in section 3.6.

10.2 PUVAME overview

In France, about 33% of roads victims are Vulnerable Road Users (VRU). In its 3^{rd} framework, the French PREDIT includes VRU Safety. The PUVAME project was created to generate solutions to avoid collisions between VRU and Bus in urban traffic. This objective will be achieved by:

- Improvement of driver's perception capabilities close to his vehicle; This objective will be achieved using a combination of offboard cameras, observing intersections or bus stops, to detect and track VRU present at intersection or bus stop, as well as onboard sensors for localisation of the bus;
- Detection and assessment of dangerous situations, analyzing position of the vehicule and of the VRU and estimating their future trajectories;
- Triggering alarms and related processes inside the vehicule;
- Integration on experimental vehicules.

Furthermore, there is a communication between the infrastructure and the bus to exchange information about the position of the bus and position of VRU present in the environment. These informations are used to compute a risk of collisions between

[1]National Research and Innovation Program for Ground Transport Systems
[2]Protection des Vulnérables par Alarmes ou Manoeuvres
[3]www.intempora.com

the bus and VRU. In case of an important risk, a warning strategy is defined to prevent the bus driver and the VRU. The project started on October 2003 and was achieved on April 2006.

The partners are:

INRIA Responsible of the following tasks: project management, providing the hardware platform, tracking using grid occupation server, system integration.

EMP-CAOR Responsible of people detection, system architecture under RTMaps and system integration.

Connex-Eurolum Responsible of Accident analysis and system validation.

Robosoft Responsible of vehicle equipment.

ProBayes Provided and supported ProBayes software.

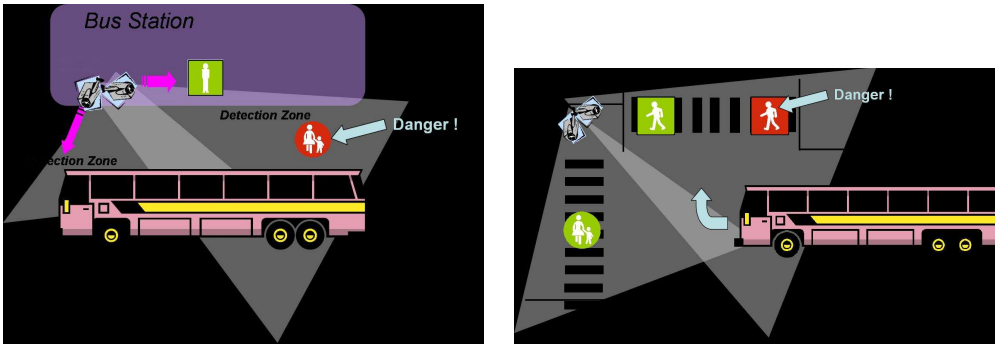
Intempora Provided and supported RTMaps.

INRETS LESCOT Responsible of the design of the warning interface for the bus.

10.3 Accident analysis and scenarios

In the scope of the project, we've analysed accidents occured in 2003 between vulnerables (pedestrians, cyclists) and buses in a French town. Three kinds of accidents arrised from this study. In 25.6% of the accidents, the vulnerable was struck while the bus was leaving or approaching a bus stop. In 38.5% of the accidents, the pedestrian was attempting to cross at an intersection when he was struck by a bus turning left or right. Finally, 33.3% of the accidents occured when the pedestrian lost balance on the sidewalk when he was running for the bus, or was struck by a lateral part of a bus or one of its rear view mirrors. It was also noticed that in all these cases, most of the impacts occurred on the right side or the front of the bus.

In the aim of reducing these kinds of accidents, we proposed 3 scenarios to reproduce the most frequent accidents' situations and find ways to overcome the lack of security. The first scenario aims at reproducing vulnerables struck while the bus arrives or leaves its bus stop (see Figure 10.1(a)). The second scenario aims at reproducing vulnerables struck at an intersection (see Figure 10.1(b)). The third scenario aims at reproducing vulnerables struck by the lateral part of the bus, surprised by the sweeping zone. In this thesis, we proposed to focus on the first two scenarios when a pedestrian is struck by the right part of the bus. In these 2 cases, it has been decided to use fixed cameras placed at the bus stop or at the road junction in order to detect potentially dangerous behaviors. As most of the right part of the bus is unseen by the driver, it is very important to give him information about the fact that a vulnerable is placed in this blind spot. The cameras will cover the entire blind zone.



(a) Bus Station (b) Cross Intersection

Figure 10.1: (a) The bus arrives or leaves its bus stop. The vulnerables situated in the blind spot near the bus are in danger because they are not seen by the driver and have a good probability to enter in collision with the bus; (b) The pedestrian crosses at an intersection when the bus turns right. The vulnerable who starts crossing the road may be unseen by the driver.

Information given by cameras will be analyzed and merged with information about the position of the bus. A collision risk estimation will be done and an interface will alert the driver about the danger. A more detailed description of the process will be done in section 10.5. Next section presents the experimental site set up at INRIA Rhône-Alpes, where these two scenarios are under test.

10.4 ParkNav platform

The experimental setup used to evaluate the PUVAME system is composed of 2 distinctive parts: the ParkView platform used to simulate an intersection or a bus stop and the Cycab vehicule used to simulate a bus.

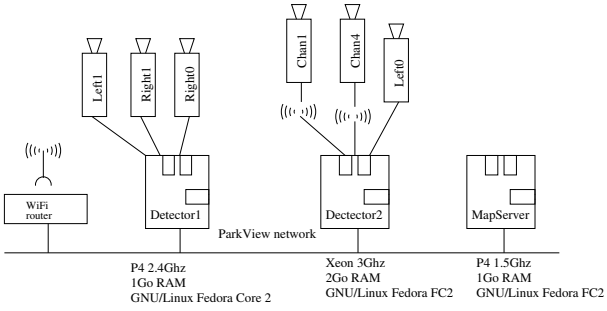
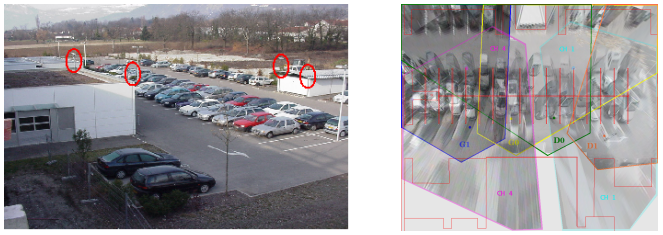


Figure 10.2: The ParkView platform hardware

10.4.1 The ParkView platform

The ParkView platform is composed of a set of six off-board analog cameras, installed in a car-park setup such as their field-of-view partially overlap (see figure 10.3), and three Linux(tm) workstations in charge of the data processing, connected by a standard Local Area Network (figure 10.2).



(a) (b)

(c) "left0" (d) "right0"

Figure 10.3: (a) Location of the cameras on the parking; (b) Field-of-view of the cameras projected on the ground; (c) and (d) View from the two cameras used

The workstations are running a specifically developed client-server software composed of three main parts, called the *map server*, the *map clients* and the *connectors* (figure 10.4).

The *map server* processes all the incoming observations, provided by the different clients, in order to maintain a global high-level representation of the environment; this is where the data fusion occurs. A single instance of the server is running.

The *map clients* connect to the server and provide the users with a graphical representation of the environment (figure 10.5); they can also process this data further and perform application-dependent tasks. For example, in a driving assistance application, the vehicle on-board computer will be running such a client specialized in estimating the collision risk.

The *connectors* are feeded with the raw sensor-data, perform the pre-processing, and send the resulting *observations* to the map server. Each of the computer connected with one or several sensors is running such a *connector*. For the application described here, all data preprocessing basically consist in detecting pedestrians. Therefore, the video stream of each camera is processed independently by a dedicated detector. The role of the detectors is to convert each incoming video frame to a set of bounding rectangles, one by target detected in the image plane (figure 10.14). The

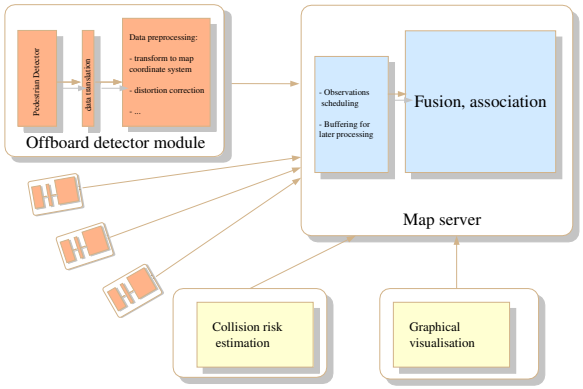


Figure 10.4: The ParkView platform software organization

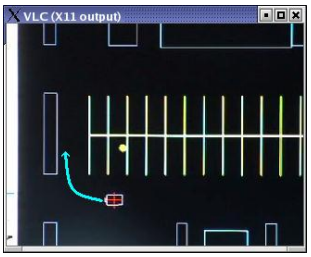


Figure 10.5: A graphical map client showing the CyCab and a pedestrian on the parking

set of rectangles detected at a given time constitutes the detector observation, and is sent to the map server.

Since the fusion system operates in a fixed coordinate system, distinct from each of the camera's local systems, a coordinate transformation must be performed. For this purpose, each of the cameras has been calibrated beforehand. The result of this calibration consists in a set of parameters:

- The intrinsic parameters contain the information about the camera optics and CCD sensor: the focal length and focal axis, the distorsion parameters,

- The extrinsic parameters consist of the homography matrix: this is the 3x3 homogeneous matrix which transforms the coordinates of an image point to the ground coordinate system.

In such a multi-sensor system, special care must be taken of proper timestamping and synchronization of the observations. This is especially true in a networked environment, where the standard TCP/IP protocol would incur its own latencies.

The ParkView platform achieves the desired effect by using a specialized transfer

protocol, building on the low-latency properties of UDP while guaranteeing in-order, synchronised delivery of the sensor observations to the server.

10.4.2 The CyCab vehicule

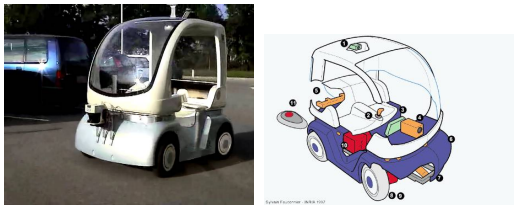


Figure 10.6: The CyCab vehicule

The CyCab (figure 10.6) has been designed to transport up to two persons in downtown areas, pedestrian malls, large industrial or amusement parks and airports, at a maximum of 30km/h speed. It has a length of 1.9 meter, a width of 1.2 meter and weights about 300 kg. It is equipped with 4 steer and drive wheels powered by four 1 kW electric motors. To control the cycab, we can manually drive it with a joystick or fully-automatic operate it. It is connected to the ParkView platform by a wireless connection: we can send it motors commands and collect odometry and sensors data: the Cycab is considered as a client of the ParkNav platform. It perceives the environment with a sick laser used to detect and avoid the obstacles.

10.5 Architecture of the system

In this section, we detail the PUVAME software architecture (figure 10.7) we choose for the intersection and bus stop. This architecture is composed of 4 main parts:

1. First of all, the images of the different offboard camera are used to estimate the position and speed of each pedestrian present in the crossroadmap;

2. The odometry and the Global Positionning System are used to determine the position and speed of the vehicule in the crossroad map;

3. Third, the position and speed of the different objects present in the intersection are used to estimate a risk of collision between the bus and each pedestrian;

4. Finally, the level of risk and the direction of the risk are sent to the Human Machine Interface (HMI) inside the bus to warn the bus driver.

In the next subsections, we detail these 4 modules.

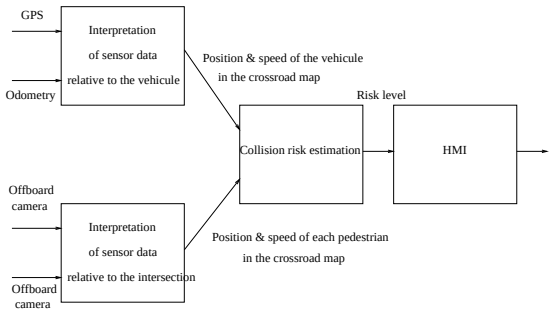


Figure 10.7: System diagram

10.5.1 Interpretation of sensor data relative to the intersection

Our objective is to have a robust perception using multi-sensor approaches to track the different pedestrians present at the intersection. The whole architecture is depicted in figure 10.8. As mentioned in section 10.4, the video camera feed is processed independently by a dedicated detector. The role of the detectors is to convert each incoming video frame to a set of bounding rectangles, one by target detected in the image plane. The set of rectangles detected at a given time constitutes the detector observations. Then, the information about the position of each VRU given by each offboard camera are merged, using an occupancy grid approach, in order to compute a better estimation of the position of each VRU. We use the output of this stage of fusion process in order to extract new observations on the VRU currently present in the environment. We use data association algorithms to update the different VRU position with extracted observations. Finally, we update the list of VRU present in the environment. The different parts of the process are detailed in the following paragraphs.

Pedestrian detector

To detect VRUs present at the intersection, a pedestrian detector subsystem is used. The detector is composed of three components: the first component consists in a foreground segmentation based on Multiple Gaussian Model as described in Section 5.3.4. The second component is a sliding window binary classifier for pedestrians using AdaBoost-based learning methods previously described in Chapter 6. The third component is a tracking algorithm using image based criteria of similarity.

Occupancy grid

The construction of the occupancy grid as a result of the fusion of the detector observations given by different cameras is detailed in [YARL06]. In this paragraph, we only give an overview of the construction of this occupancy grid.

Occupancy grid is a generic framework for multi-sensor fusion and modelling of the environment. It has been introduced by Elfes and Moravec [Elf89] at the end

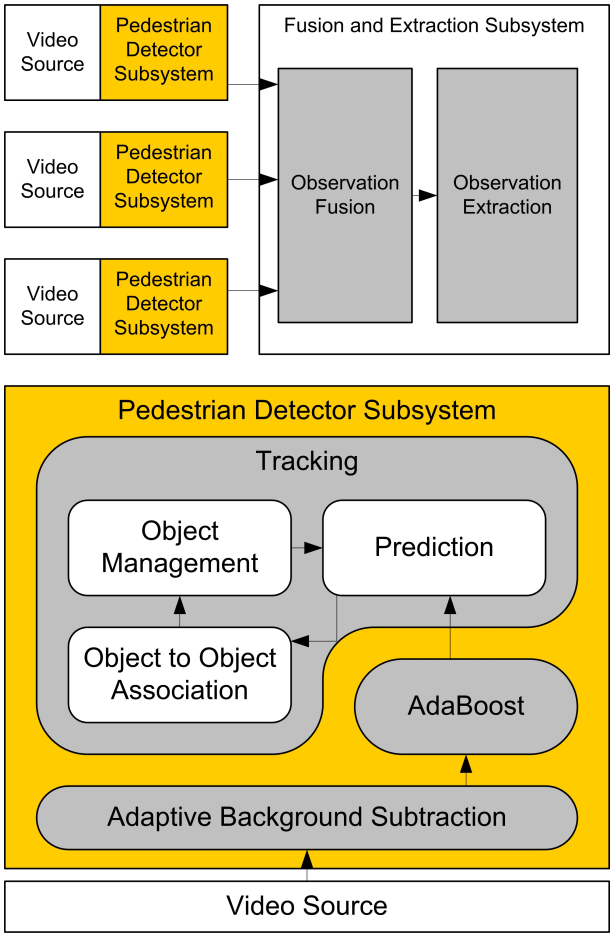
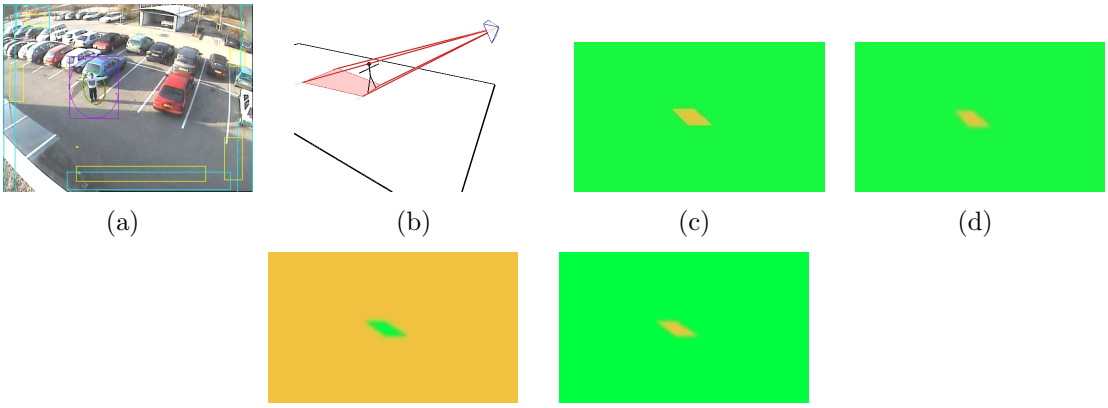


Figure 10.8: Architecture of the pedestrians tracker

of the 1980s. An occupancy grid is a stochastic tesselated representation of spatial information that maintains probabilistic estimates of the occupancy state of each cell in a lattice. The main advantage of this approach is the ability to integrate several sensors in the same framework taking the inherent uncertainty of each sensor reading into account, in opposite to the Geometric Paradigm whose method is to categorize the world features into a set of geometric primitives [CSW03]. The alternative that OGs offer is a regular sampling of the space occupancy, that is a very generic system of space representation when no knowledge about the shapes of the environment is available. On the contrary of a feature based environment model, the only requirement for an OG building is a bayesian sensor model for each cell of the grid and each sensor. This sensor model is the description of the probabilistic relation that links sensor measurement to space state, that OG necessitates to make the sensor integration.

In [YARL06], we propose there two different sensor models that are suitable for different purposes, but which underline the genericity of the occupancy grid approach. The problem is that motion detectors give information in the image space and that

we search to have knowledge in the ground plan. We solve this problem projecting the bounding box in the ground plan using some hypothesis: in the first model, we mainly suppose that the ground is a plan, all the VRU stand on the ground and the complete VRU is visible for the camera. The second model is more general as we consider that a VRU could be partially hidden but has a maximum height of 3 meters.

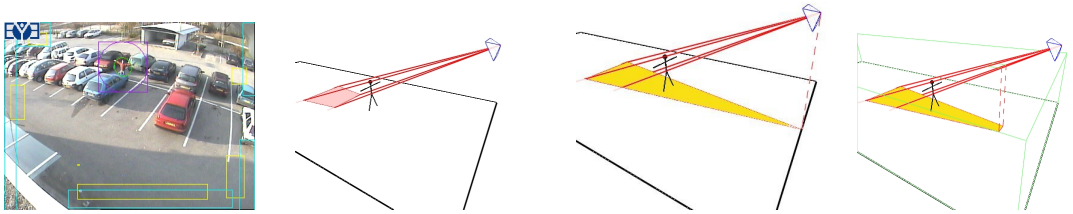


(a) (b) (c) (d) (e) (f)

Figure 10.9: (a) An image of a moving object acquired by one of the offboard video cameras and the associated bounding box found by the detector. (b) The occulted zone as the intersection of the viewing cone associated with the bounding box and the ground plan. (c) The associated ground image produce by the system. (d) Ground image after gaussian convolution with a support size of 7 pixels. (e) Probability of the ground image pixel value, knowing that the pixel corresponds to an empty cell: $P(Z|\text{emp})$ for each cell. (f) Probability of the ground image pixel value, knowing that the pixel corresponds to an occupied cell: $P(Z|\text{occ})$ for each cell.

In both of the models we first search to segment the ground plan in three types of region: occupied, occulted and free zones using the bounding boxes informations. Then we introduce an uncertainty management, using a gaussian convolution, to deal with the position errors in the detector. Finally, we convert this information into probability distributions. Figure 10.9 illustrates the whole construction of the first model and figure 10.10 shows the first phase for the second model.

Figure 10.11 shows experiments (ie, the resulting occupancy grid) with the first model. The first model is precise, but only when its hypothesis holds. In such cases this model will be the most suitable for position estimation. With the second model, the position uncertainty allows to surround the real position of the detected object, such that with other viewing points or other sensors, like laser range-finders or radar it is possible to obtain a good hull of the ground object occupation. Thanks to the uncertainty, this last model will never give wrong information about the emptiness of an area, which is a guarantee for safety applications.



(a) (b) (c) (d)

Figure 10.10: (a) Moving object whom the contact points with the ground are occulted. (b) The intersection of the viewing cone associated with the bounding box and the ground plan which is far from the position of the object. (c) Projection of the entire view cone on the ground in yellow. (d) Projection of the part of view cone that fits the object height hypothesis (in green).

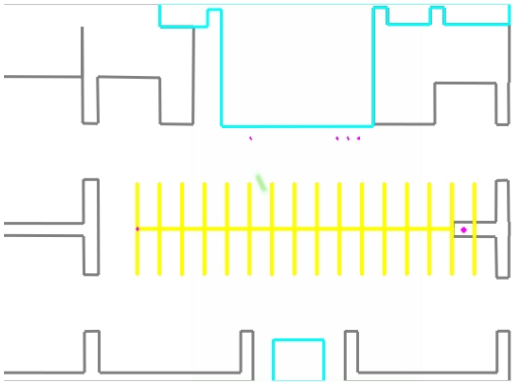


Figure 10.11: The resulting probability that the cells are occupied after the inference process.

Object extraction

Based on the occupancy grid, we extract the objects in the grid: we extract moving obstacles by first identifying the moving area. We proceed by differencing two consecutive occupancy grids in time, as in [BA04].

Prediction

To track VRUs, a Kalman filter [Kal60] is used for each VRU present in the environment. The speed needed for the prediction phase of each Kalman filter is computed making a difference between the actual and the previous position.

Track to track association

To reestimate the position of each VRU using a Kalman filter, we first need to associate the observations of VRUs extracted from the occupancy grid to the predicted position. As there could be at most one observation associated to a given predicted position, we use a gating procedure which is enough for correct assignments. The association is also useful to manage the list of VRUs present in the environment, as described in the next paragraph.

Track management

Each VRU is tagged with a specific ID and its position in the environment. At the beginning of the process, the list of VRUs present in the environment is empty. The result of the association phase is used to update this list. Several cases could appear:

1. An observation is associated to a VRU: the reestimated position of this VRU is computed with a Kalman filter the predicted position and this observation;
2. A VRU has no observation associated to itself: the reestimated position of this VRU is equal to the predicted position;
3. An observation is not associated to any VRU: a new temporary VRU ID is created and its position is initialized as the value of the observation. To avoid to create VRU corresponding to false alarms, the temporary VRU is only confirmed (ie, becomes a definitive VRU) if it is seen during 3 consecutive instants.

As we are using offboard cameras observing always the same environment, to delete a VRU of the list, 2 conditions are needed: it has to be unseen (ie, no observation has been associated to himself) since at least the last 5 instants and its position has to be outside of the intersection.

10.5.2 Interpretation of sensor data relative to the vehicule

The goal of this module [KDdlF+04] is to compute a precise position of the bus at the intersection. This position is computed using an extended Kalman filter with odometry data for predicting the position of the vehicule and a GPS to estimate its position. The localisation is based on a precise datation of the data to minimize the effect of latency.

10.5.3 Collision Risk Estimation

The risk of collision is computed with the Closest Point of Approach and the Time to Closest Point of Approach method. The Closest Point of Approach is the point where two moving objects are at a minimum distance. To compute this distance, we suppose that these two moving objects are moving at a constant speed. So at each time, we are able to compute the distance between these two objects. To know when

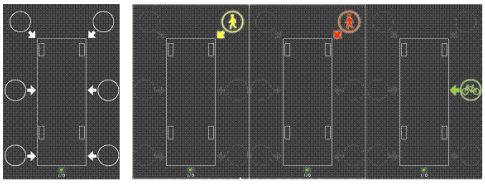


Figure 10.12: interface (left) and different warning functions

Figure 10.13: interface (left) and different warning functions

this distance is minimal, we search the instant where the derivative of the square of this distance is null. This instant is named the Time to Closest Point of Approach and the respective distance is named the Closest Point of Approach.

In our application, at each time, we compute the Time to Closest Point of Approach and the respective distance (ie, Closest Point of Approach) between each VRU and the bus. If this distance is below a given threshold in less than 5 seconds, we compute the orientation of the VRU relative to the bus and generate an alarm for the HMI.

10.5.4 Warning interface

The interface (figure 10.12) is made of two parts. One is the bus outline (center of the interface), and the other is the possible position of vulnerables (the 6 circles). These circles can be fullfilled by warning pictograms. These pictograms show the place where a vulnerable could be struck by the bus : in front, middle, back, left or right side of the bus. The representation needs to be very simple in order to be

Figure 10.14: (left) two pedestrians are crossing the road. They are detected by our system. The Cycab is arriving turning right. (center) The system estimates that there is a high probability of collision between the pedestrian starting crossing the road. (right) It alerts the driver with a signal on the interface

understood immediately by the driver. Moreover, two speakers are added in order to warn rapidly the driver about the danger (figure 10.13).

10.6 Experimental results

We reproduced the scenario shown (see Figure 10.1(b)) in our experimental site. The bus is a Cycab, two pedestrians crossing the road are detected by our system. One is in danger because it has a high probability to be struck by the Cycab. The driver is alerted by the interface that a pedestrian is potentially in danger (see Figure 10.14).

Part V

Conclusion and future work

Chapter 11

Conclusion

Contents

11.1 Overview

In the present thesis we have presented a generic framework for intelligent video analysis. We have focused on the development of several stages within this framework: adaptive background subtraction and visual object detection. We have developed basic implementations for the rest of the stages in this framework (tracking module). We have accelerated some algorithms using a low cost solution. We have prototyped an application for urban zone safety. This application is built on the top of the given medium level algorithms and the low cost architecture.

This work is composed of four main parts: the development of medium level algorithms specialized in background subtraction and object detection, the usage of the graphics hardware as a target for some algorithms, the design of the algorithms for the execution on the graphics hardware and finally the integration of the different algorithms within an application.

We have worked with state of the art algorithms for background subtraction and visual object detection. We have implemented several approaches for background subtraction. We have compared the performance of these methods and we have enhanced their accuracy using high level feedback.

We have developed *LibAdaBoost*, a generic software library for machine learning. We have extended this library to implement a boosting framework. We have adopted a plugin oriented architecture for this library. We have extended this framework for the visual domain and implemented the state of the art features as well as we have developed new features. With *LibAdaBoost* it has been easy to work with the different actors of the boosting framework. Thus, we have easily enhanced the genetic

algorithm (previously developed by Y. Abramson) as a weak learner. *LibAdaBoost* makes it possible to objectively compare the different features (performance and accuracy). For the best of our knowledge, *LibAdaBoost* is the first machine learning library specialized in boosting for the visual domain.

The low cost solution is based on using the graphics hardware for computer vision purpose. This has been a good choice according to the performance profile of the application. The graphics hardware has a data-parallelism oriented architecture. This architecture is interfaced through a streaming model of computation. We have introduced a streaming design of the visual object detection algorithm. This design was used to map the algorithm on the graphics hardware. For the best of our knowledge, we were the first to implement a visual object detector on the graphics hardware.

Finally, we integrated those algorithms within a large application in an industrial context. The PUVAME project was a good opportunity to evaluate those algorithms and to experiment with the integration issues. The results still are at a proof of concept stage. We aimed at validating the system on some elementary scenarios where we have done some simplifications. We have got promising results, accuracy speaking as well as performance speaking.

11.2 Future work

Beside the various contributions in the present thesis, the complete framework for intelligent video analysis still is non perfect. Many improvements could be introduced at several levels. In this section we will present a non exhaustive list of improvements as future work issues:

- The generic framework for intelligent video analysis as presented in Chapter 4 still is uncomplete. We have explored in our research some stages of this framework and not all the stages. The exploration of the other stages (object tracking, action recognition, semantic description, personal identification and fusion of multiple cameras) makes the application range wider. Thus, we can consider more advanced applications based on fusion of multiple sensors as well as a recognition system for controlling high security areas.

- The adaptive background subtraction algorithms presented in Chapter 5 are not perfect. They have the advantage of being generic. This means that they have not special preprocessing for a given installation. Otherwise, those algorithms require fusion with other algorithms (visual object detection or motion detection) to perform a good accuracy. Recently, new approaches for background subtraction are proposed. The new approaches are based on machine learning techniques. In their work, Parag et al. [PEM06] used boosting for selecting the features to represent a given scene background at the pixel level. We propose to explore this possibility using the boosting framework *LibAdaBoost* (cf. Chapter 6).

- *LibAdaBoost* is a complete machine learning framework. *LibAdaBoost* provides a boosting framework. This boosting framework is extended for the visual object detection domain. It could be extended for other domains (as described above, for pixel classification domain). It has two major points: extensibility and robustness. This library could be used as a reference for any boosting related algorithms.

- In our research, we explored the GPGPU for accelerating computer vision algorithms. We reached promising performance results for the visual object detection. The new graphics hardware generation starting by the G8 series (NVIDIA) offers a complete solution for general purpose computing. NVIDIA proposes CUDA as software solution for GPGPU. We propose to continue the development on the CUDA framework to accelerate the different algorithms introduced in this thesis.

- The PUVAME project is a promising project for improving pedestrian safety in urban zones. The pedestrian detection and tracking system was validated on special scenarios, and still has lackings to be fully operational in real condition scenes. Thus, more advanced research must continue on the object detection part, as well as on the object tracking part.

- Video surveillance applications for large scale installations are migrating to third generation. The usage of smart cameras as distributed agents imposes new architecture and new problematics related to the cooperation and the computing distribution as well as the fusion of information from multiple cameras. The implementation of computer vision algorithms on these cameras is a special task, the onboard embedded processors have generally special architectures. The mapping of the algorithms to smart cameras could be guided by a codesign activity which aims to develop new cores for accelerating computer vision tasks using special hardware. We believe that second generation video surveillance systems still have a long life, and we still have to develop systems upon second generation architectures. Thus, the methodology used within this thesis still has to be optimized and has a promising market.

Chapter 12

French conclusion

Dans le présent manuscrit, on a présenté une chaîne de traitement générique pour l'analyse vidéo intelligente. On a exploré plusieurs implémentations relatives aux différents étages dans cette chaîne: modèles adaptatifs du fond d'une scène dynamique à camèra fixe ainsi que la détection visuelle d'objets par apprentissage automatique en utilisant le boosting. On a pu développer des implémentations de base pour le reste des étapes de la chaîne de traitement. On a accéléré une partie de ces algorithmes en utilisant des architectures â faible coût basées sur les cartes graphiques programmables. On a prototypé une application pour améliorer la sécurité des piétons dans les zones urbaines près des arrêts du bus.

Ce travail est composé de quatres parties principales: le développement d'algorithmes spécialisés dans la segmentation du fond et la détection visuelle d'objets, l'utilisation des cartes graphiques pour l'accélération de certains algorithmes de vision, la modélisation des algorithmes pour l'exécustion sur des processeurs graphiques comme cible et finalement, l'intégration des diffèrents algorithmes dans le cadre d'une application.

On a travaillé sur des algorithmes relevant de l'état de l'art sur la ségmentation du fond et sur la détection visuelle d'objets. On a implémenté plusieurs mèthodes pour la segmentation du fond. Ces mèthodes sont basées sur le modèle adaptatif. On a étudié la performance de chacune de ces mèthodes et on a essayé d'améliorer la robustesse en utilisant le retour d'informations de haut niveau.

On a développé *LibAdaBoost*, une librairie pour l'apprentissage automatique implémentant la technique du boosting en utilisant un modèle générique des données. Cette librairie est étendue pour suporter le domaine visuel. On a implémenté les classifieurs faibles publiés dans l'état de l'art ainsi que nos propres classifieurs faibles. *LibAdaBoost* a facilité l'utilisation de la technique du boosting pour le domaine visuel. On a pu améliorer facilement l'algorithme génétique (développé par Y. Abramson). *LibAdaBoost* fournitdes fonctionnalités qui permettent une comparaison objective des diffèrents classifierus faibles ainsi que les algorithmes d'apprentissage et tout autre degrès de liberté. A nos connaissances *LibAdaBoost* est la premiére librarie d'apprentissage automatique qui fournit une implémentation du boosting qui est robuste, extensible et qui supporte un modèle de données générique.

La solution à faible coût est basée sur les processeurs graphiques qui seront ex-

ploités pour l'exécution des algorithmes de vision. Il s'est avèré que cette solution est une solution adaptée à ce type d'algorithmes. Le processeur graphique est basé sur une architecture qui favorise le parallélisme de données. Les applications ciblés sur ce processeur sont modélisés en utilisant un modèle de calcul orienté flux de données. Ce modèle de calcul est basé sur deux élements structureaux qui sont les flux de données *streams* et les noyaux de calcul *kernels*. A nos connaissances, nous sommes les premiers qui ont implémenté le premier detecteur visuel d'objets basé sur le boosting sur un processeur graphique.

Finalement, on a intégré ces algorithmes dans une large application dans un contexte industriel. Le projet PUVAME était une bonne oportunité pour évaluer ces algorithmes. On a pu aussi expérimenté la phase d'intégration avec les autres sous systèmes développés par les autres partenaires. Les résultats restent au niveau de l'étude de faisabilité. On a fait recours á une validation ponctuelle des algorithmes sur des scénarios simplifiés. On a obtenu des résultats encourageant, au niveau de la robustesse et des performances.

Pourtant les diffèrentes contributions à travers ce travail, la totalité de la chaine de traitement reste inparfaite. Plusieurs améliorations pourront avoir lieu. Une liste non exhaustive de potentielle améliorations pourra être la suivante:

- La chaîne de traitement pour l'analyse vidéo intelligente présentée dans Chapitre 4 est incompl'éte. On a exploré quelques étapes de cette chaîne, et il nous reste encors plusieurs comme le suivi d'objets, la reconnaisance des objets, la description sémantique du contenu, l'analyse des actions et la collaboration entre plusieurs caméras sous forme de fusion de données. Cependant, cette exploration des autres étapes permettra d'élargir le spectre des applications qui pourraient être utilisés dans le controle d'accés dans des zones à fortes contraintes de sécurité.

- Les algorithmes de segmentation du fond adaptatifs présentés dans Chapitre 5 ne sont pas parfaits. Ils ont l'avantage d'être génériques. Cela veut dire que ces algorithmes ne demandent pas un paramétrage préalable en fonction de la scéne surveillée, mais ils ont besoins d'être fusionnés avec d'autres algorithmes pour arriver à une meilleure performance. Une nouvelle approche basée sur l'apprentissage automatique en utilisant le boosting au niveau du pixel est utilisée par Parag et al. [PEM06]. On propose d'esplorer cette solution. Ceci est facilité par notre librairie de boosting *LibAdaBoost* (cf. Chapitre 6).

- *LibAdaBoost* est une librairie complète dédiée pour l'apprentissage automatique. Cette librairie implémente la technique de boosting d'une maniére générique. Cette technique est étendu au domaine visuel. Elle pourra être étendue pour d'autres types de données (comme décrit ci dessus au niveau des pixels pour la segmentation du fond). Cette librairie a deux points avantageux: l'extensibilité et la robustesse. Cette librairie servira comme environnement de référence pour la recherche sur la technique du boosting.

- On a exploré l'utilisation des processeurs graphiques pour l'accélération des

algorithmes de vision qui reprèsentent un parallélisme de données fonctionnel. On a obtenu des résultats meilleurs que le processeur traditionnel. Les nouvelles cartes graphiques chez NVIDIA de la série G8 sont programmables moyennant une solution logiciele révolutionnaire qui s'appelle CUDA. Cette solution logicielle permettant la programmation des processeurs graphiques avec le langage C ANSI sans passer par les couches de rendu 3D comme OpenGL et DirectX couplés aux langages de shader comme Cg. On propose de continuer le développement sur les cartes graphique sur les processeurs G8 avec la solution CUDA.

- Le projet PUVAME est prometeur pour l'amélioration de la sécurité des piétons dans les zones urbaines. Malgrès la validation sur des scénarios élémentaires, les algorithmes de détection et de suivi devront être améliorés pour obtenir un système operationnel dans les conditions réelles.

- Les systèmes de vidéo surveillance de grande envergure sont en migration vers la troisiéme génération. L'utilisation des caméra intelligentes comme des agents distribués impose une nouvelle méthode de pensée et drelèves de nouvelle problématique au niveau de la coopération entre ces agents et de la distribution du calcul entre eux. L'écriture des algorithmes de vision pour ces caméras comme cibles est tout un art. Les processeurs embarqués sur ces caméras ont des architectures spéciales. Et pour le moment il n'y a pas des normes qui unifient ces aspects architecturaux. On pense que malgrès cette migration, la deuxiéme génération des systèmes de vidéo surveillance a un temps de vie assez long qui justifie la poursuite des dévelopements dédiés pour ces systèmes. La méthodologie utilisée dans le cadre de cette thése devra être améliorée. Auccune crainte budgétaire, le marché est prometteur.

Part VI

Appendices

Appendix A

Hello World GPGPU

Listing A.1: Hello world GPGPU OpenGL/GLSL

```
1  //------------------------------------------------------------------------------
2  //                                www.GPGPU.org
3  //                                 Sample Code
4  //------------------------------------------------------------------------------
5  // Copyright (c) 2004 Mark J. Harris and GPGPU.org
6  // Copyright (c) 2004 3Dlabs Inc. Ltd.
7  //------------------------------------------------------------------------------
8  // This software is provided 'as-is', without any express or implied
9  // warranty. In no event will the authors be held liable for any
10 // damages arising from the use of this software.
11 //
12 // Permission is granted to anyone to use this software for any
13 // purpose, including commercial applications, and to alter it and
14 // redistribute it freely, subject to the following restrictions:
15 //
16 // 1. The origin of this software must not be misrepresented; you
17 //    must not claim that you wrote the original software. If you use
18 //    this software in a product, an acknowledgment in the product
19 //    documentation would be appreciated but is not required.
20 //
21 // 2. Altered source versions must be plainly marked as such, and
22 //    must not be misrepresented as being the original software.
23 //
24 // 3. This notice may not be removed or altered from any source
25 //    distribution.
26 //
27 //------------------------------------------------------------------------------
28 // Author: Mark Harris (harrism@gpgpu.org) - original helloGPGPU
29 // Author: Mike Weiblen (mike.weiblen@3dlabs.com) - GLSL version
30 //------------------------------------------------------------------------------
31 // GPGPU Lesson 0: "helloGPGPU_GLSL" (a GLSL version of "helloGPGPU")
32 //------------------------------------------------------------------------------
33 //
34 // GPGPU CONCEPTS Introduced:
35 //
```

```
36 //      1.) Texture = Array
37 //      2.) Fragment Program = Computational Kernel.
38 //      3.) One-to-one Pixel to Texel Mapping:
39 //          a) Data-Dimensioned Viewport, and
40 //          b) Orthographic Projection.
41 //      4.) Viewport-Sized Quad = Data Stream Generator.
42 //      5.) Copy To Texture = feedback.
43 //
44 //      For details of each of these concepts, see the explanations in the
45 //      inline "GPGPU CONCEPT" comments in the code below.
46 //
47 // APPLICATION Demonstrated: A simple post-process edge detection filter.
48 //
49 //-----------------------------------------------------------------------
50 // Notes regarding this "helloGPGPU_GLSL" source code:
51 //
52 // This example was derived from the original "helloGPGPU.cpp" v1.0.1
53 // by Mark J. Harris.  It demonstrates the same simple post-process edge
54 // detection filter, but instead implemented using the OpenGL Shading Language
55 // (also known as "GLSL"), an extension of the OpenGL v1.5 specification.
56 // Because the GLSL compiler is an integral part of a vendor's OpenGL driver,
57 // no additional GLSL development tools are required.
58 // This example was developed/tested on 3Dlabs Wildcat Realizm.
59 //
60 // I intentionally minimized changes to the structure of the original code
61 // to support a side-by-side comparison of the implementations.
62 //
63 // Thanks to Mark for making the original helloGPGPU example available!
64 //
65 // -- Mike Weiblen, May 2004
66 //
67 //
68 // [MJH:]
69 // This example has also been tested on NVIDIA GeForce FX and GeForce 6800 GPUs.
70 //
71 // Note that the example requires glew.h and glew32s.lib, available at
72 // http://glew.sourceforge.net.
73 //
74 // Thanks to Mike for modifying the example.  I have actually changed my
75 // original code to match; for example the inline Cg code instead of an
76 // external file.
77 //
78 // -- Mark Harris, June 2004
79 //
80 // References:
81 //    http://gpgpu.sourceforge.net/
82 //    http://glew.sourceforge.net/
83 //    http://www.xmission.com/~nate/glut.html
84 //-----------------------------------------------------------------------
85
86 #include <stdio.h>
87 #include <assert.h>
```

```
88 #include <stdlib.h>
89 #define GLEW_STATIC 1
90 #include <GL/glew.h>
91 #include <GL/glut.h>
92
93 // forward declarations
94 class HelloGPGPU;
95 void reshape(int w, int h);
96
97 // globals
98 HelloGPGPU  *g_pHello;
99
100
101 // This shader performs a 9-tap Laplacian edge detection filter.
102 // (converted from the separate "edges.cg" file to embedded GLSL string)
103 static const char *edgeFragSource = {
104 "uniform sampler2D texUnit;"
105 "void main(void)"
106 "{"
107 "    const float offset = 1.0 / 512.0;"
108 "    vec2 texCoord = gl_TexCoord[0].xy;"
109 "    vec4 c  = texture2D(texUnit, texCoord);"
110 "    vec4 bl = texture2D(texUnit, texCoord + vec2(-offset, -offset));"
111 "    vec4 l  = texture2D(texUnit, texCoord + vec2(-offset,     0.0));"
112 "    vec4 tl = texture2D(texUnit, texCoord + vec2(-offset,  offset));"
113 "    vec4 t  = texture2D(texUnit, texCoord + vec2(    0.0,  offset));"
114 "    vec4 ur = texture2D(texUnit, texCoord + vec2( offset,  offset));"
115 "    vec4 r  = texture2D(texUnit, texCoord + vec2( offset,     0.0));"
116 "    vec4 br = texture2D(texUnit, texCoord + vec2( offset,  offset));"
117 "    vec4 b  = texture2D(texUnit, texCoord + vec2(    0.0, -offset));"
118 "    gl_FragColor = 8.0 * (c + -0.125 * (bl + l + tl + t + ur + r + br + b));"
119 "}"
120 };
121
122 // This class encapsulates all of the GPGPU functionality of the example.
123 class HelloGPGPU
124 {
125 public: // methods
126     HelloGPGPU(int w, int h)
127     : _rAngle(0),
128       _iWidth(w),
129       _iHeight(h)
130     {
131         // Create a simple 2D texture.  This example does not use
132         // render to texture -- it just copies from the framebuffer to the
133         // texture.
134
135         // GPGPU CONCEPT 1: Texture = Array.
136         // Textures are the GPGPU equivalent of arrays in standard
137         // computation. Here we allocate a texture large enough to fit our
138         // data (which is arbitrary in this example).
139         glGenTextures(1, &_iTexture);
```

```
140         glBindTexture(GL_TEXTURE_2D, _iTexture);
141         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
142         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
143         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
144         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
145         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, _iWidth, _iHeight,
146                      0, GL_RGB, GL_FLOAT, 0);
147
148         // GPGPU CONCEPT 2: Fragment Program = Computational Kernel.
149         // A fragment program can be thought of as a small computational
150         // kernel that is applied in parallel to many fragments
151         // simultaneously.  Here we load a kernel that performs an edge
152         // detection filter on an image.
153
154         _programObject = glCreateProgramObjectARB();
155
156         // Create the edge detection fragment program
157         _fragmentShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
158         glShaderSourceARB(_fragmentShader, 1, &edgeFragSource, NULL);
159         glCompileShaderARB(_fragmentShader);
160         glAttachObjectARB(_programObject, _fragmentShader);
161
162         // Link the shader into a complete GLSL program.
163         glLinkProgramARB(_programObject);
164         GLint progLinkSuccess;
165         glGetObjectParameterivARB(_programObject, GL_OBJECT_LINK_STATUS_ARB,
166                 &progLinkSuccess);
167         if (!progLinkSuccess)
168         {
169             fprintf(stderr, "Filter shader could not be linked\n");
170             exit(1);
171         }
172
173         // Get location of the sampler uniform
174         _texUnit = glGetUniformLocationARB(_programObject, "texUnit");
175     }
176
177         ~HelloGPGPU()
178     {
179     }
180
181     // This method updates the texture by rendering the geometry (a teapot
182     // and 3 rotating tori) and copying the image to a texture.
183     // It then renders a second pass using the texture as input to an edge
184     // detection filter.  It copies the results of the filter to the texture.
185     // The texture is used in HelloGPGPU::display() for displaying the
186     // results.
187     void update()
188     {
189         _rAngle += 0.5f;
190
191         // store the window viewport dimensions so we can reset them,
```

```
192         // and set the viewport to the dimensions of our texture
193         int vp[4];
194         glGetIntegerv(GL_VIEWPORT, vp);
195 
196         // GPGPU CONCEPT 3a: One-to-one Pixel to Texel Mapping: A Data-
197         //                   Dimensioned Viewport.
198         // We need a one-to-one mapping of pixels to texels in order to
199         // ensure every element of our texture is processed. By setting our
200         // viewport to the dimensions of our destination texture and drawing
201         // a screen-sized quad (see below), we ensure that every pixel of our
202         // texel is generated and processed in the fragment program.
203         glViewport(0, 0, _iWidth, _iHeight);
204 
205         // Render a teapot and 3 tori
206         glClear(GL_COLOR_BUFFER_BIT);
207         glMatrixMode(GL_MODELVIEW);
208         glPushMatrix();
209         glRotatef(-_rAngle, 0, 1, 0.25);
210         glutSolidTeapot(0.5);
211         glPopMatrix();
212         glPushMatrix();
213         glRotatef(2.1 * _rAngle, 1, 0.5, 0);
214         glutSolidTorus(0.05, 0.9, 64, 64);
215         glPopMatrix();
216         glPushMatrix();
217         glRotatef(-1.5 * _rAngle, 0, 1, 0.5);
218         glutSolidTorus(0.05, 0.9, 64, 64);
219         glPopMatrix();
220         glPushMatrix();
221         glRotatef(1.78 * _rAngle, 0.5, 0, 1);
222         glutSolidTorus(0.05, 0.9, 64, 64);
223         glPopMatrix();
224 
225         // copy the results to the texture
226         glBindTexture(GL_TEXTURE_2D, _iTexture);
227         glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, _iWidth, _iHeight);
228 
229 
230         // run the edge detection filter over the geometry texture
231         // Activate the edge detection filter program
232         glUseProgramObjectARB(_programObject);
233 
234         // identify the bound texture unit as input to the filter
235         glUniform1iARB(_texUnit, 0);
236 
237         // GPGPU CONCEPT 4: Viewport-Sized Quad = Data Stream Generator.
238         // In order to execute fragment programs, we need to generate pixels.
239         // Drawing a quad the size of our viewport (see above) generates a
240         // fragment for every pixel of our destination texture. Each fragment
241         // is processed identically by the fragment program. Notice that in
242         // the reshape() function, below, we have set the frustum to
243         // orthographic, and the frustum dimensions to [-1,1]. Thus, our
```

```
244         // viewport-sized quad vertices are at [-1,-1], [1,-1], [1,1], and
245         // [-1,1]: the corners of the viewport.
246         glBegin(GL_QUADS);
247         {
248             glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
249             glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
250             glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
251             glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
252         }
253         glEnd();
254 
255         // disable the filter
256         glUseProgramObjectARB(0);
257 
258         // GPGPU CONCEPT 5: Copy To Texture (CTT) = Feedback.
259         // We have just invoked our computation (edge detection) by applying
260         // a fragment program to a viewport-sized quad. The results are now
261         // in the frame buffer. To store them, we copy the data from the
262         // frame buffer to a texture.  This can then be fed back as input
263         // for display (in this case) or more computation (see
264         // more advanced samples.)
265 
266         // update the texture again, this time with the filtered scene
267         glBindTexture(GL_TEXTURE_2D, _iTexture);
268         glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, _iWidth, _iHeight);
269 
270         // restore the stored viewport dimensions
271         glViewport(vp[0], vp[1], vp[2], vp[3]);
272     }
273 
274     void display()
275     {
276         // Bind the filtered texture
277         glBindTexture(GL_TEXTURE_2D, _iTexture);
278         glEnable(GL_TEXTURE_2D);
279 
280         // render a full-screen quad textured with the results of our
281         // computation.  Note that this is not part of the computation: this
282         // is only the visualization of the results.
283         glBegin(GL_QUADS);
284         {
285             glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
286             glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
287             glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
288             glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
289         }
290         glEnd();
291 
292         glDisable(GL_TEXTURE_2D);
293     }
294 
295 protected: // data
```

```
296     int             _iWidth, _iHeight; // The dimensions of our array
297     float           _rAngle;           // used for animation
298
299     unsigned int    _iTexture;         // The texture used as a data array
300
301     GLhandleARB     _programObject;    // the program used to update
302     GLhandleARB     _fragmentShader;
303
304     GLint           _texUnit;          // a parameter to the fragment program
305 };
306
307 // GLUT idle function
308 void idle()
309 {
310     glutPostRedisplay();
311 }
312
313 // GLUT display function
314 void display()
315 {
316     g_pHello->update();  // update the scene and run the edge detect filter
317     g_pHello->display(); // display the results
318     glutSwapBuffers();
319 }
320
321 // GLUT reshape function
322 void reshape(int w, int h)
323 {
324     if (h == 0) h = 1;
325
326     glViewport(0, 0, w, h);
327
328     // GPGPU CONCEPT 3b: One-to-one Pixel to Texel Mapping: An Orthographic
329     //                   Projection.
330     // This code sets the projection matrix to orthographic with a range of
331     // [-1,1] in the X and Y dimensions. This allows a trivial mapping of
332     // pixels to texels.
333     glMatrixMode(GL_PROJECTION);
334     glLoadIdentity();
335     gluOrtho2D(-1, 1, -1, 1);
336     glMatrixMode(GL_MODELVIEW);
337     glLoadIdentity();
338 }
339
340 // Called at startup
341 void initialize()
342 {
343     // Initialize the "OpenGL Extension Wrangler" library
344     glewInit();
345
346     // Ensure we have the necessary OpenGL Shading Language extensions.
347     if (glewGetExtension("GL_ARB_fragment_shader")      != GL_TRUE ||
```

```
348         glewGetExtension("GL_ARB_vertex_shader")         != GL_TRUE ||
349         glewGetExtension("GL_ARB_shader_objects")        != GL_TRUE ||
350         glewGetExtension("GL_ARB_shading_language_100") != GL_TRUE)
351     {
352         fprintf(stderr, "Driver does not support OpenGL Shading Language\n");
353         exit(1);
354     }
355 
356     // Create the example object
357     g_pHello = new HelloGPGPU(512, 512);
358 }
359 
360 // The main function
361 int main()
362 {
363     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
364     glutInitWindowSize(512, 512);
365     glutCreateWindow("Hello, GPGPU! (GLSL version)");
366 
367     glutIdleFunc(idle);
368     glutDisplayFunc(display);
369     glutReshapeFunc(reshape);
370 
371     initialize();
372 
373     glutMainLoop();
374     return 0;
375 }
```

Appendix B

Hello World Brook

Listing B.1: BR file

```
/*-----------------------------------------------------------------------
                        GPGPU - Computer Vision
                Adaptive Background Subtraction on Graphics Hardware
-------------------------------------------------------------------------
 Copyright (c) 2005 - 2006 Hicham Ghorayeb
-------------------------------------------------------------------------
 This software is provided 'as-is', without any express or implied
 warranty. In no event will the authors be held liable for any
 damages arising from the use of this software.

 Permission is granted to anyone to use this software for any
 purpose, including commercial applications, and to alter it and
 redistribute it freely, subject to the following restrictions:

 1. The origin of this software must not be misrepresented; you
    must not claim that you wrote the original software. If you use
    this software in a product, an acknowledgment in the product
    documentation would be appreciated but is not required.

 2. Altered source versions must be plainly marked as such, and
    must not be misrepresented as being the original software.

 3. This notice may not be removed or altered from any source
    distribution.

-------------------------------------------------------------------------
 Author: Hicham Ghorayeb (hicham.ghorayeb@ensmp.fr)
-------------------------------------------------------------------------
 Notes regarding this "GPU based Adaptive Background Subtraction" source code:


 Note that the example requires OpenCV and Brook

 -- Hicham Ghorayeb, April 2006

```

```
36  References:
37      http://www.gpgpu.org/
38  ----------------------------------------------------------------------------*/
39
40  #include <stdio.h>
41  #include <stdio.h>
42  #include <stdlib.h>
43  #include <string.h>
44  #include "../bbs.h"
45  #include <cv.h>
46  #include "TimeUtils/time_tools.h"
47  #define BBS_ALGO_STR    "BASIC_BACKGROUND_SUBTRACTION"
48  #define BBS_ALGO_STR_D "BASIC_BACKGROUND_SUBTRACTION:Download"
49  #define BBS_ALGO_STR_K_CLASSIFY "BASIC_BACKGROUND_SUBTRACTION:Kernel:Classify"
50  #define BBS_ALGO_STR_K_UPDATE    "BASIC_BACKGROUND_SUBTRACTION:Kernel:Update"
51  #define BBS_ALGO_STR_K_INIT "BASIC_BACKGROUND_SUBTRACTION:Kernel:Init"
52  #define BBS_ALGO_STR_U "BASIC_BACKGROUND_SUBTRACTION:Upload"
53
54  /***********************************************
55   * STREAMS DECLARATION
56   ***********************************************/
57  int insizex  =  BBS_FRAME_WIDTH;
58  int insizey  = BBS_FRAME_HEIGHT;
59  int outsizex =  BBS_FRAME_WIDTH;
60  int outsizey = BBS_FRAME_HEIGHT;
61
62  iter float2 it<outsizex,outsizey> =
63       iter( float2(0.0f, 0.0f), float2(insizey, insizex) );
64
65  float3 backgroundModel<insizex,insizey>;
66  float3 inputFrame<insizex,insizey>;
67  float foregroundMask<insizex,insizey>;
68
69  float      alpha = BBS_ALPHA;
70  float threshold = BBS_THRESHOLD;
71
72  /***********************************************
73   * KERNELS DECLARATION
74   ***********************************************/
75  kernel
76  void br_kernel_bbs_init(
77       float3 iImg[][],
78       iter float2 it<>,
79       out float3 oModel<>)
80  {
81          oModel = iImg[it];
82  }
83
84  kernel
85  void br_kernel_bbs_classify(
86       float3 iImg[][],
87       float3 iModel[][],
```

```
88      iter float2 it<>,
89      float threshold,
90      out float oMask<>)
91  {
92          float3 regAbsDiff;
93          float3 inPixels =   iImg[it];
94          float3 inModel  = iModel[it];
95          regAbsDiff = abs(inPixels - inModel);
96
97          if ( all(step(threshold, regAbsDiff) ) )
98                  oMask = 255.0f;
99          else
100                 oMask =   0.0f;
101 }
102
103 kernel
104 void br_kernel_bbs_update(
105     float3 iImg[][],
106     float3 iModel[][],
107     iter float2 it<>,
108     float alpha,
109     out float3 oModel<>)
110 {
111         float3 inPixels =   iImg[it];
112         float3 inModel  = iModel[it];
113         float3 a = float3(alpha, alpha, alpha);
114
115         oModel = a * inPixels + (1-a) * inModel;
116 }
117
118 /*********************************************
119  * BRIDGES TO KERNELS
120  *********************************************/
121 void bbs_read_input_streams(float *image)
122 {
123         INIT_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR);
124         START_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_D);
125         streamRead(inputFrame, image);
126         STOP_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_D);
127 }
128
129 void bbs_write_output_streams(float *image)
130 {
131         INIT_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR);
132         START_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_U);
133         streamWrite(foregroundMask, image);
134         STOP_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_U);
135 }
136
137 void run_bbsInit()
138 {
139         INIT_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR);
```

```
140         START_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_K_INIT);
141         br_kernel_bbs_init(inputFrame, it, backgroundModel);
142         STOP_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_K_INIT);
143 }
144
145 void run_bbsClassify()
146 {
147         INIT_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR);
148         START_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_K_CLASSIFY);
149         br_kernel_bbs_classify(
150               inputFrame,
151               backgroundModel,
152               it,
153               threshold,
154               foregroundMask);
155         STOP_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_K_CLASSIFY);
156         }
157
158 void run_bbsUpdate()
159 {
160         INIT_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR);
161         START_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_K_UPDATE);
162         br_kernel_bbs_update(
163                inputFrame,
164                backgroundModel,
165                it,
166                alpha,
167                backgroundModel);
168         STOP_BRMM_PERFORMANCE_MEASURE(BBS_ALGO_STR_K_UPDATE);
169 }
```

Listing B.2: GPU subsystem

```
1  //------------------------------------------------------------------------
2  //                        GPGPU - Computer Vision
3  //           Adaptive Background Subtraction on Graphics Hardware
4  //------------------------------------------------------------------------
5  // Copyright (c) 2005 - 2006 Hicham Ghorayeb
6  //------------------------------------------------------------------------
7  // This software is provided 'as-is', without any express or implied
8  // warranty. In no event will the authors be held liable for any
9  // damages arising from the use of this software.
10 //
11 // Permission is granted to anyone to use this software for any
12 // purpose, including commercial applications, and to alter it and
13 // redistribute it freely, subject to the following restrictions:
14 //
15 // 1. The origin of this software must not be misrepresented; you
16 //    must not claim that you wrote the original software. If you use
17 //    this software in a product, an acknowledgment in the product
18 //    documentation would be appreciated but is not required.
19 //
20 // 2. Altered source versions must be plainly marked as such, and
```

```
21 //     must not be misrepresented as being the original software.
22 //
23 // 3. This notice may not be removed or altered from any source
24 //     distribution.
25 //
26 //------------------------------------------------------------------------------
27 // Author: Hicham Ghorayeb (hicham.ghorayeb@ensmp.fr)
28 //------------------------------------------------------------------------------
29 // Notes regarding this "GPU based Adaptive Background Subtraction" source code:
30 //
31 //
32 // Note that the example requires OpenCV and Brook
33 //
34 // -- Hicham Ghorayeb, April 2006
35 //
36 // References:
37 //     http://www.gpgpu.org/
38 //------------------------------------------------------------------------------
39
40 #include "gpu_subsystem.h"
41
42 #include "bbs.h"
43
44         // Init Subsystem
45 int gpu_init_subsystem()
46 {
47         // Allocates Float Buffer
48         return 0;
49 }
50
51         // Release Subsystem
52 void gpu_release_subsystem()
53 {
54         // Release Float Buffer
55 }
56
57         // Initialization of the Background Image
58 void gpu_InitBackground(struct _IplImage *image)
59 {
60         assert((image->width == BBS_FRAME_WIDTH)  &&
61          (image->height == BBS_FRAME_HEIGHT)      &&
62           (image->nChannels == 3));
63
64         float* buffer  = NULL;
65
66         buffer = (float*)malloc(
67         image->width * image->height * image->nChannels * sizeof(float));
68
69         // Init Buffer from IplImage
70         for(int i=0; i<image->height; i++){
71                 for(int j=0; j<image->width; j++){
72                         buffer[ i * image->width * 3 + j * 3 + 0] =
```

```
73                          (float)image->imageData[i * image->widthStep+j* 3+ 0];
74                          buffer[ i * image->width * 3 + j * 3 + 1] =
75                          (float)image->imageData[i * image->widthStep+j* 3+ 1];
76                          buffer[ i * image->width * 3 + j * 3 + 2] =
77                          (float)image->imageData[i * image->widthStep+j* 3+ 2];
78                  }
79          }
80
81          bbs_read_input_streams(buffer);
82          run_bbsInit();
83
84          free( buffer );
85  }
86
87          // Call The Algorithm to classify pixels into Foreground and background
88  void  gpu_ClassifyImage(struct _IplImage *image)
89  {
90          assert((image->width == BBS_FRAME_WIDTH)  &&
91           (image->height == BBS_FRAME_HEIGHT) &&
92           (image->nChannels == 3));
93
94          float* buffer  = NULL;
95
96          buffer = (float*)malloc(
97          image->width * image->height * image->nChannels * sizeof(float));
98
99          // Init Buffer from IplImage
100         for(int i=0; i<image->height; i++){
101                 for(int j=0; j<image->width; j++){
102                         buffer[ i * image->width * 3 + j * 3 + 0] =
103                          (float)image->imageData[i * image->widthStep+j* 3+ 0];
104                         buffer[ i * image->width * 3 + j * 3 + 1] =
105                          (float)image->imageData[i * image->widthStep+j* 3+ 1];
106                         buffer[ i * image->width * 3 + j * 3 + 2] =
107                          (float)image->imageData[i * image->widthStep+j* 3+ 2];
108                 }
109         }
110
111         bbs_read_input_streams(buffer);
112         run_bbsClassify();
113
114         free( buffer );
115 }
116
117         // Retrieve the result of background/foreground segmentation
118 void gpu_RetrieveResult(struct _IplImage *image)
119 {
120         assert((image->width == BBS_FRAME_WIDTH)  &&
121          (image->height == BBS_FRAME_HEIGHT) &&
122           (image->nChannels == 1));
123
124         float* buffer  = NULL;
```

```
125
126         buffer = (float*)malloc(
127         image->width * image->height * image->nChannels * sizeof(float));
128
129         bbs_write_output_streams(buffer);
130
131         // Update IplImage from the Buffer
132         for(int i=0; i<image->height; i++){
133                 for(int j=0; j<image->width; j++){
134                         image->imageData[i*image->widthStep+j+0] =
135                          ((buffer[i * image->width + j + 0] > 0.0f )? 255:0);
136                 }
137         }
138
139         free( buffer );
140 }
141
142         // Update the Background Model
143 void gpu_UpdateBackgroundModel()
144 {
145         run_bbsUpdate();
146 }
```

Listing B.3: CPU subsystem

```
1  //-------------------------------------------------------------------------
2  //                        GPGPU - Computer Vision
3  //            Adaptive Background Subtraction on Graphics Hardware
4  //-------------------------------------------------------------------------
5  // Copyright (c) 2005 - 2006 Hicham Ghorayeb
6  //-------------------------------------------------------------------------
7  // This software is provided 'as-is', without any express or implied
8  // warranty. In no event will the authors be held liable for any
9  // damages arising from the use of this software.
10 //
11 // Permission is granted to anyone to use this software for any
12 // purpose, including commercial applications, and to alter it and
13 // redistribute it freely, subject to the following restrictions:
14 //
15 // 1. The origin of this software must not be misrepresented; you
16 //    must not claim that you wrote the original software. If you use
17 //    this software in a product, an acknowledgment in the product
18 //    documentation would be appreciated but is not required.
19 //
20 // 2. Altered source versions must be plainly marked as such, and
21 //    must not be misrepresented as being the original software.
22 //
23 // 3. This notice may not be removed or altered from any source
24 //    distribution.
25 //
26 //-------------------------------------------------------------------------
27 // Author: Hicham Ghorayeb (hicham.ghorayeb@ensmp.fr)
28 //-------------------------------------------------------------------------
```

```
29 // Notes regarding this "GPU based Adaptive Background Subtraction" source code:
30 //
31 //
32 // Note that the example requires OpenCV and Brook
33 //
34 // -- Hicham Ghorayeb, April 2006
35 //
36 // References:
37 //    http://www.gpgpu.org/
38 //-------------------------------------------------------------------------
39 #include "cpu_subsystem.h"
40
41 int cpu_init_subsystem()
42 {
43         return 0;
44 }
45
46 void cpu_release_subsystem()
47 {
48 }
```

Listing B.4: Main function

```
1  //-------------------------------------------------------------------------
2  //                        GPGPU - Computer Vision
3  //           Adaptive Background Subtraction on Graphics Hardware
4  //-------------------------------------------------------------------------
5  // Copyright (c) 2005 - 2006 Hicham Ghorayeb
6  //-------------------------------------------------------------------------
7  // This software is provided 'as-is', without any express or implied
8  // warranty. In no event will the authors be held liable for any
9  // damages arising from the use of this software.
10 //
11 // Permission is granted to anyone to use this software for any
12 // purpose, including commercial applications, and to alter it and
13 // redistribute it freely, subject to the following restrictions:
14 //
15 // 1. The origin of this software must not be misrepresented; you
16 //    must not claim that you wrote the original software. If you use
17 //    this software in a product, an acknowledgment in the product
18 //    documentation would be appreciated but is not required.
19 //
20 // 2. Altered source versions must be plainly marked as such, and
21 //    must not be misrepresented as being the original software.
22 //
23 // 3. This notice may not be removed or altered from any source
24 //    distribution.
25 //
26 //-------------------------------------------------------------------------
27 // Author: Hicham Ghorayeb (hicham.ghorayeb@ensmp.fr)
28 //-------------------------------------------------------------------------
29 // Notes regarding this "GPU based Adaptive Background Subtraction" source code:
30 //
```

```
31 //
32 // Note that the example requires OpenCV and Brook
33 //
34 // -- Hicham Ghorayeb, April 2006
35 //
36 // References:
37 //    http://www.gpgpu.org/
38 //-------------------------------------------------------------------------
39
40 #include <cv.h>
41 #include <highgui.h>
42
43 #include <stdio.h>
44 #include <stdlib.h>
45 #include <string.h>
46 #include <assert.h>
47 #include <math.h>
48 #include <float.h>
49 #include <limits.h>
50 #include <time.h>
51 #include <ctype.h>
52
53 #include "BasicBackgroundSubtraction.h"
54 #include "cpu_subsystem.h"
55 #include "gpu_subsystem.h"
56
57 void do_processing( IplImage *frame);
58
59 int main( int argc, char** argv )
60 {
61         /*****************************************
62         * Declarations
63         *****************************************/
64     CvCapture* capture = NULL;
65     IplImage *frame, *frame_copy = NULL;
66
67         /*****************************************
68         * Init Video Source
69         *****************************************/
70
71 #ifdef _USE_WEBCAM_
72         capture = cvCaptureFromCAM( 0 );
73 #else
74         capture = cvCaptureFromAVI( argv[1] );
75 #endif
76
77         /*****************************************
78         * Init Video Sink
79         *****************************************/
80         cvNamedWindow(       "video", 1 );
81         cvNamedWindow(  "foreground", 1 );
82
```

```
83          /*****************************************
84          * Main Video Analysis Loop
85          *****************************************/
86          if( capture ){
87      for(;;){
88          if( !cvGrabFrame( capture ))
89            break;
90          frame = cvRetrieveFrame( capture );
91          if( !frame )
92            break;
93          if( !frame_copy ){
94                      printf(
95                        "Input Video Dimensions : Width = %d, Height = %d\n",
96                        frame->width,frame->height);
97                      frame_copy = cvCreateImage(
98                       cvSize(frame->width,frame->height),
99          IPL_DEPTH_8U, frame->nChannels );
100       }
101
102       if( frame->origin == IPL_ORIGIN_TL )
103        cvCopy( frame, frame_copy, 0 );
104       else
105        cvFlip( frame, frame_copy, 0 );
106
107       do_processing( frame_copy );
108
109                   if( cvWaitKey( 10 ) >= 0 )
110        break;
111       }
112
113       cvReleaseImage( &frame_copy );
114       cvReleaseCapture( &capture );
115       }
116
117       cvDestroyWindow("video");
118       cvDestroyWindow("foreground");
119       return 0;
120   }
121
122   void do_processing( IplImage *frame)
123   {
124     static bool first_frame = true;
125
126           // Show Input Image
127           cvShowImage( "video", frame );
128
129           IplImage *foreground = NULL;
130
131           foreground = cvCreateImage(
132           cvSize(frame->width,frame->height), IPL_DEPTH_8U, 1);
133
134           if( first_frame )
```

```
135         {
136         gpu_InitBackground( frame );
137                 first_frame = false;
138                 return;
139         }
140
141         gpu_ClassifyImage( frame );
142
143         gpu_RetrieveResult( foreground );
144
145         gpu_UpdateBackgroundModel();
146
147         // Show Result Of Processing
148         cvShowImage( "foreground", foreground);
149
150     cvReleaseImage( &foreground );
151 }
```

Appendix C

Hello World CUDA

Listing C.1: Kernels file

```
1  /*
2   * Copyright 1993-2007 NVIDIA Corporation.  All rights reserved.
3   *
4   * NOTICE TO USER:
5   *
6   * This source code is subject to NVIDIA ownership rights under U.S. and
7   * international Copyright laws.  Users and possessors of this source code
8   * are hereby granted a nonexclusive, royalty-free license to use this code
9   * in individual and commercial software.
10  *
11  * NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
12  * CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
13  * IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
14  * REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
15  * MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
16  * IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
17  * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
18  * OF USE, DATA OR PROFITS,  WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
19  * OR OTHER TORTIOUS ACTION,  ARISING OUT OF OR IN CONNECTION WITH THE USE
20  * OR PERFORMANCE OF THIS SOURCE CODE.
21  *
22  * U.S. Government End Users.  This source code is a "commercial item" as
23  * that term is defined at  48 C.F.R. 2.101 (OCT 1995), consisting  of
24  * "commercial computer  software"  and "commercial computer software
25  * documentation" as such terms are  used in 48 C.F.R. 12.212 (SEPT 1995)
26  * and is provided to the U.S. Government only as a commercial end item.
27  * Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
28  * 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
29  * source code with only those rights set forth herein.
30  *
31  * Any use of this source code in individual and commercial software must
32  * include, in the user documentation and internal comments to the code,
33  * the above Disclaimer and U.S. Government End Users Notice.
34  */
35
```

```
36 #include <stdio.h>
37 #include <stdlib.h>
38 #include "SobelFilter_kernels.h"
39
40 #define SV 0.003921f
41 #define IV 255.f
42
43 // Texture reference for reading image
44 texture<unsigned char, 2> tex;
45 extern __shared__ unsigned char LocalBlock[];
46
47 #define Radius 1
48
49 #ifdef FIXED_BLOCKWIDTH
50 #define BlockWidth 80
51 #define SharedPitch 384
52 #endif
53
54 __device__ unsigned char
55 ComputeSobel(unsigned char ul, // upper left
56              unsigned char um, // upper middle
57              unsigned char ur, // upper right
58              unsigned char ml, // middle left
59              unsigned char mm, // middle (unused)
60              unsigned char mr, // middle right
61              unsigned char ll, // lower left
62              unsigned char lm, // lower middle
63              unsigned char lr, // lower right
64              float fScale )
65 {
66     short Horz = ur + 2*mr + lr - ul - 2*ml - ll;
67     short Vert = ul + 2*um + ur - ll - 2*lm - lr;
68     short Sum = (short) (fScale*(abs(Horz)+abs(Vert)));
69     if ( Sum < 0 ) return 0; else if ( Sum > 0xff ) return 0xff;
70     return (unsigned char) Sum;
71 }
72
73 __global__ void
74 SobelShared( uchar4 *pSobelOriginal, unsigned short SobelPitch,
75 #ifndef FIXED_BLOCKWIDTH
76              short BlockWidth, short SharedPitch,
77 #endif
78              short w, short h, float fScale )
79 {
80     short u = 4*blockIdx.x*BlockWidth;
81     short v = blockIdx.y*blockDim.y + threadIdx.y;
82     short ib;
83
84     int SharedIdx = threadIdx.y * SharedPitch;
85
86     for ( ib = threadIdx.x; ib < BlockWidth+2*Radius; ib += blockDim.x ) {
87         LocalBlock[SharedIdx+4*ib+0] = tex2D( tex,
```

```
88              (float) (u+4*ib-Radius+0), (float) (v-Radius) );
89          LocalBlock[SharedIdx+4*ib+1] = tex2D( tex,
90              (float) (u+4*ib-Radius+1), (float) (v-Radius) );
91          LocalBlock[SharedIdx+4*ib+2] = tex2D( tex,
92              (float) (u+4*ib-Radius+2), (float) (v-Radius) );
93          LocalBlock[SharedIdx+4*ib+3] = tex2D( tex,
94              (float) (u+4*ib-Radius+3), (float) (v-Radius) );
95      }
96      if ( threadIdx.y < Radius*2 ) {
97          //
98          // copy trailing Radius*2 rows of pixels into shared
99          //
100         SharedIdx = (blockDim.y+threadIdx.y) * SharedPitch;
101         for ( ib = threadIdx.x; ib < BlockWidth+2*Radius; ib += blockDim.x ) {
102             LocalBlock[SharedIdx+4*ib+0] = tex2D( tex,
103                 (float) (u+4*ib-Radius+0), (float) (v+blockDim.y-Radius) );
104             LocalBlock[SharedIdx+4*ib+1] = tex2D( tex,
105                 (float) (u+4*ib-Radius+1), (float) (v+blockDim.y-Radius) );
106             LocalBlock[SharedIdx+4*ib+2] = tex2D( tex,
107                 (float) (u+4*ib-Radius+2), (float) (v+blockDim.y-Radius) );
108             LocalBlock[SharedIdx+4*ib+3] = tex2D( tex,
109                 (float) (u+4*ib-Radius+3), (float) (v+blockDim.y-Radius) );
110         }
111     }
112
113     __syncthreads();
114
115     u >>= 2;    // index as uchar4 from here
116     uchar4 *pSobel = (uchar4 *) (((char *) pSobelOriginal)+v*SobelPitch);
117     SharedIdx = threadIdx.y * SharedPitch;
118
119     for ( ib = threadIdx.x; ib < BlockWidth; ib += blockDim.x ) {
120
121         unsigned char pix00 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+0];
122         unsigned char pix01 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+1];
123         unsigned char pix02 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+2];
124         unsigned char pix10 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+0];
125         unsigned char pix11 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+1];
126         unsigned char pix12 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+2];
127         unsigned char pix20 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+0];
128         unsigned char pix21 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+1];
129         unsigned char pix22 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+2];
130
131         uchar4 out;
132
133         out.x = ComputeSobel(pix00, pix01, pix02,
134                              pix10, pix11, pix12,
135                              pix20, pix21, pix22, fScale );
136
137         pix00 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+3];
138         pix10 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+3];
139         pix20 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+3];
```

```
140         out.y = ComputeSobel(pix01, pix02, pix00,
141                              pix11, pix12, pix10,
142                              pix21, pix22, pix20, fScale );
143
144         pix01 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+4];
145         pix11 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+4];
146         pix21 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+4];
147         out.z = ComputeSobel( pix02, pix00, pix01,
148                               pix12, pix10, pix11,
149                               pix22, pix20, pix21, fScale );
150
151         pix02 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+5];
152         pix12 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+5];
153         pix22 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+5];
154         out.w = ComputeSobel( pix00, pix01, pix02,
155                               pix10, pix11, pix12,
156                               pix20, pix21, pix22, fScale );
157         if ( u+ib < w/4 && v < h ) {
158             pSobel[u+ib] = out;
159         }
160     }
161
162     __syncthreads();
163 }
164
165 __global__ void
166 SobelCopyImage( unsigned char *pSobelOriginal, unsigned int Pitch,
167                 int w, int h )
168 {
169     unsigned char *pSobel =
170       (unsigned char *) (((char *) pSobelOriginal)+blockIdx.x*Pitch);
171     for ( int i = threadIdx.x; i < w; i += blockDim.x ) {
172         pSobel[i] = tex2D( tex, (float) i, (float) blockIdx.x );
173     }
174 }
175
176 __global__ void
177 SobelTex( unsigned char *pSobelOriginal, unsigned int Pitch,
178           int w, int h, float fScale )
179 {
180     unsigned char *pSobel =
181       (unsigned char *) (((char *) pSobelOriginal)+blockIdx.x*Pitch);
182     for ( int i = threadIdx.x; i < w; i += blockDim.x ) {
183         unsigned char pix00 = tex2D( tex, (float) i-1, (float) blockIdx.x-1 );
184         unsigned char pix01 = tex2D( tex, (float) i+0, (float) blockIdx.x-1 );
185         unsigned char pix02 = tex2D( tex, (float) i+1, (float) blockIdx.x-1 );
186         unsigned char pix10 = tex2D( tex, (float) i-1, (float) blockIdx.x+0 );
187         unsigned char pix11 = tex2D( tex, (float) i+0, (float) blockIdx.x+0 );
188         unsigned char pix12 = tex2D( tex, (float) i+1, (float) blockIdx.x+0 );
189         unsigned char pix20 = tex2D( tex, (float) i-1, (float) blockIdx.x+1 );
190         unsigned char pix21 = tex2D( tex, (float) i+0, (float) blockIdx.x+1 );
191         unsigned char pix22 = tex2D( tex, (float) i+1, (float) blockIdx.x+1 );
```

```
192         pSobel[i] = ComputeSobel(pix00, pix01, pix02,
193                                  pix10, pix11, pix12,
194                                  pix20, pix21, pix22, fScale );
195     }
196 }
```

Listing C.2: Main file

```
1  /*
2   * Copyright 1993-2007 NVIDIA Corporation.  All rights reserved.
3   *
4   * NOTICE TO USER:
5   *
6   * This source code is subject to NVIDIA ownership rights under U.S. and
7   * international Copyright laws.  Users and possessors of this source code
8   * are hereby granted a nonexclusive, royalty-free license to use this code
9   * in individual and commercial software.
10  *
11  * NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
12  * CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
13  * IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
14  * REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
15  * MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
16  * IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
17  * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
18  * OF USE, DATA OR PROFITS,  WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
19  * OR OTHER TORTIOUS ACTION,  ARISING OUT OF OR IN CONNECTION WITH THE USE
20  * OR PERFORMANCE OF THIS SOURCE CODE.
21  *
22  * U.S. Government End Users.  This source code is a "commercial item" as
23  * that term is defined at  48 C.F.R. 2.101 (OCT 1995), consisting  of
24  * "commercial computer  software"  and "commercial computer software
25  * documentation" as such terms are  used in 48 C.F.R. 12.212 (SEPT 1995)
26  * and is provided to the U.S. Government only as a commercial end item.
27  * Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
28  * 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
29  * source code with only those rights set forth herein.
30  *
31  * Any use of this source code in individual and commercial software must
32  * include, in the user documentation and internal comments to the code,
33  * the above Disclaimer and U.S. Government End Users Notice.
34  */
35
36 #include <stdlib.h>
37 #include <stdio.h>
38 #include <string.h>
39 #include <GL/glew.h>
40 #include <GL/glut.h>
41 #include <cutil.h>
42 #include <cutil_interop.h>
43 #include <cuda_gl_interop.h>
44
45 #include "SobelFilter_kernels.cu"
```

```
46
47 //
48 // Cuda example code that implements the Sobel edge detection
49 // filter. This code works for 8-bit monochrome images.
50 //
51 // Use the '-' and '=' keys to change the scale factor.
52 //
53 // Other keys:
54 // I: display image
55 // T: display Sobel edge detection (computed solely with texture)
56 // S: display Sobel edge detection (computed with texture and shared memory)
57
58 void cleanup(void);
59 void initializeData(int w, int h);
60
61 static int wWidth   = 512; // Window width
62 static int wHeight  = 512; // Window height
63 static int imWidth  = 0;   // Image width
64 static int imHeight = 0;   // Image height
65
66 static int fpsCount = 0;        // FPS count for averaging
67 static int fpsLimit = 1;        // FPS limit for sampling
68 unsigned int timer;
69
70 // Display Data
71 static GLuint pbuffer = 0;     // Front and back CA buffers
72 static GLuint texid = 0;       // Texture for display
73 Pixel *pixels = NULL;          // Image pixel data on the host
74 float imageScale = 1.f;        // Image exposure
75 enum SobelDisplayMode g_SobelDisplayMode;
76
77 static cudaArray *array = NULL;
78
79
80 #define OFFSET(i) ((char *)NULL + (i))
81
82 // Wrapper for the __global__ call that sets up the texture and threads
83 void
84 sobelFilter(Pixel *odata, int iw, int ih, enum SobelDisplayMode mode,
85             float fScale) {
86
87     CUDA_SAFE_CALL(cudaBindTextureToArray(tex, array));
88
89     switch ( mode ) {
90         case  SOBELDISPLAY_IMAGE:
91             SobelCopyImage<<<ih, 384>>>(odata, iw, iw, ih );
92             break;
93         case SOBELDISPLAY_SOBELTEX:
94             SobelTex<<<ih, 384>>>(odata, iw, iw, ih, fScale );
95             break;
96         case SOBELDISPLAY_SOBELSHARED:
97         {
```

```
98              dim3 threads(16,4);
99  #ifndef FIXED_BLOCKWIDTH
100                     int BlockWidth = 80; // must be divisible by 16 for coalescing
101 #endif
102             dim3 blocks = dim3(iw/(4*BlockWidth)+
103                                (0!=iw %(4*BlockWidth)),
104                                ih/threads.y+
105                                (0!=ih%threads.y));
106                             int SharedPitch = ~0x3f&(4*(BlockWidth+2*Radius)+0x3f);
107                             int sharedMem = SharedPitch*(threads.y+2*Radius);
108
109                             // for the shared kernel, width must be divisible by 4
110                             iw &= ~3;
111
112                             SobelShared<<<blocks, threads, sharedMem>>>
113                             ((uchar4 *) odata,
114                             iw,
115 #ifndef FIXED_BLOCKWIDTH
116             BlockWidth, SharedPitch,
117 #endif
118             iw, ih, fScale );
119         }
120         break;
121     }
122
123     CUDA_SAFE_CALL(cudaUnbindTexture(tex));
124 }
125
126 void display(void) {
127
128     // Sobel operation
129     Pixel *data = NULL;
130     CUDA_SAFE_CALL(cudaGLMapBufferObject((void**)&data, pbuffer));
131     CUT_SAFE_CALL(cutStartTimer(timer));
132     sobelFilter(data, imWidth, imHeight, g_SobelDisplayMode, imageScale );
133     CUT_SAFE_CALL(cutStopTimer(timer));
134     CUDA_SAFE_CALL(cudaGLUnmapBufferObject(pbuffer));
135
136     glClear(GL_COLOR_BUFFER_BIT);
137
138     glBindTexture(GL_TEXTURE_2D, texid);
139     glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbuffer);
140     glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, imWidth, imHeight,
141                     GL_LUMINANCE, GL_UNSIGNED_BYTE, OFFSET(0));
142     glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
143
144     glDisable(GL_DEPTH_TEST);
145     glEnable(GL_TEXTURE_2D);
146     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
147     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
148     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
149     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

```
150
151     glBegin(GL_QUADS);
152     glVertex2f(0, 0); glTexCoord2f(0, 0);
153     glVertex2f(0, 1); glTexCoord2f(1, 0);
154     glVertex2f(1, 1); glTexCoord2f(1, 1);
155     glVertex2f(1, 0); glTexCoord2f(0, 1);
156     glEnd();
157     glBindTexture(GL_TEXTURE_2D, 0);
158
159     glutSwapBuffers();
160
161     fpsCount++;
162     if (fpsCount == fpsLimit) {
163         char fps[256];
164         float ifps = 1.f / (cutGetAverageTimerValue(timer) / 1000.f);
165         sprintf(fps, "Cuda Edge Detection: %3.1f fps", ifps);
166         glutSetWindowTitle(fps);
167         fpsCount = 0;
168         fpsLimit = (int)max(ifps, 1.f);
169         CUT_SAFE_CALL(cutResetTimer(timer));
170     }
171
172     glutPostRedisplay();
173 }
174
175 void idle(void) {
176     glutPostRedisplay();
177 }
178
179 void keyboard( unsigned char key, int x, int y) {
180     switch( key) {
181         case 27:
182         exit (0);
183                 break;
184                 case '-':
185                         imageScale -= 0.1f;
186                 break;
187                 case '=':
188                         imageScale += 0.1f;
189                 break;
190                 case 'i': case 'I':
191                     g_SobelDisplayMode = SOBELDISPLAY_IMAGE;
192                 break;
193                 case 's': case 'S':
194                     g_SobelDisplayMode = SOBELDISPLAY_SOBELSHARED;
195                 break;
196                 case 't': case 'T':
197                     g_SobelDisplayMode = SOBELDISPLAY_SOBELTEX;
198         default: break;
199     }
200     glutPostRedisplay();
201 }
```

```
202
203 void reshape(int x, int y) {
204     glViewport(0, 0, x, y);
205     glMatrixMode(GL_PROJECTION);
206     glLoadIdentity();
207     glOrtho(0, 1, 0, 1, 0, 1);
208     glMatrixMode(GL_MODELVIEW);
209     glLoadIdentity();
210     glutPostRedisplay();
211 }
212
213 void setupTexture(int iw, int ih, Pixel *data)
214 {
215     cudaChannelFormatDesc desc = cudaCreateChannelDesc<unsigned char>();
216     CUDA_SAFE_CALL(cudaMallocArray(&array, &desc, iw, ih));
217     CUDA_SAFE_CALL(cudaMemcpyToArray(array, 0, 0,
218     data, sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice));
219 }
220
221 void deleteTexture(void)
222 {
223     CUDA_SAFE_CALL(cudaFreeArray(array));
224 }
225
226 void cleanup(void) {
227     CUDA_SAFE_CALL(cudaGLUnregisterBufferObject(pbuffer));
228
229     glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
230     glDeleteBuffers(1, &pbuffer);
231     glDeleteTextures(1, &texid);
232
233     deleteTexture();
234
235     CUT_SAFE_CALL(cutDeleteTimer(timer));
236 }
237
238 void initializeData(char *file) {
239     GLint bsize;
240     unsigned int w, h;
241     if (cutLoadPGMub(file, &pixels, &w, &h) != CUTTrue) {
242         printf("Failed to load image file: %s\n", file);
243         exit(-1);
244     }
245     imWidth = (int)w; imHeight = (int)h;
246     setupTexture(imWidth, imHeight, pixels);
247     memset(pixels, 0x0, sizeof(Pixel) * imWidth * imHeight);
248
249     glGenBuffers(1, &pbuffer);
250     glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbuffer);
251     glBufferData(GL_PIXEL_UNPACK_BUFFER,
252                  sizeof(Pixel) * imWidth * imHeight,
253                  pixels, GL_STREAM_DRAW);
```

```
254
255     glGetBufferParameteriv(GL_PIXEL_UNPACK_BUFFER, GL_BUFFER_SIZE, &bsize);
256     if (bsize != (sizeof(Pixel) * imWidth * imHeight)) {
257         printf("Buffer object (%d) has incorrect size (%d).\n", pbuffer, bsize);
258         exit(-1);
259     }
260
261     glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
262     CUDA_SAFE_CALL(cudaGLRegisterBufferObject(pbuffer));
263
264     glGenTextures(1, &texid);
265     glBindTexture(GL_TEXTURE_2D, texid);
266     glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, imWidth, imHeight,
267                  0, GL_LUMINANCE, GL_UNSIGNED_BYTE, NULL);
268     glBindTexture(GL_TEXTURE_2D, 0);
269
270     glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
271     glPixelStorei(GL_PACK_ALIGNMENT, 1);
272 }
273
274 void loadDefaultImage( char* loc_exec) {
275
276     printf("Reading image lena.pgm.\n");
277     const char* image_filename = "lena.pgm";
278     char* image_path = cutFindFilePath(image_filename, loc_exec);
279     if (image_path == 0) {
280        printf( "Reading image failed.\n");
281        exit(EXIT_FAILURE);
282     }
283     initializeData( image_path);
284     cutFree( image_path);
285 }
286
287 int main(int argc, char** argv) {
288
289 #ifndef __DEVICE_EMULATION__
290     if( CUTFalse == isInteropSupported()) {
291         return 1;
292     }
293 #endif // __DEVICE_EMULATION__
294
295     CUT_DEVICE_INIT();
296
297     CUT_SAFE_CALL(cutCreateTimer(&timer));
298     CUT_SAFE_CALL(cutResetTimer(timer));
299
300     glutInit( &argc, argv);
301     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
302     glutInitWindowSize(wWidth, wHeight);
303     glutCreateWindow("Cuda Edge Detection");
304     glutDisplayFunc(display);
305     glutKeyboardFunc(keyboard);
```

```
306     glutReshapeFunc(reshape);
307     glutIdleFunc(idle);
308 
309     glewInit();
310     if (!glewIsSupported("GL_VERSION_2_0 GL_VERSION_1_5 GL_ARB_vertex_buffer_object 
311         fprintf(stderr, "Required OpenGL extensions missing.");
312         exit(-1);
313     }
314 
315     if (argc > 1) {
316         // test if in regression mode
317         if( 0 != strcmp( "-noprompt", argv[1])) {
318            initializeData(argv[1]);
319         }
320         else {
321             loadDefaultImage( argv[0]);
322         }
323     } else {
324         loadDefaultImage( argv[0]);
325     }
326     printf("I: display image\n");
327     printf("T: display Sobel edge detection (computed with tex)\n");
328     printf("S: display Sobel edge detection (computed with tex+shared memory)\n");
329     printf("Use the '-' and '=' keys to change the brightness.\n");
330     fflush(stdout);
331     atexit(cleanup);
332     glutMainLoop();
333 
334     return 0;
335 }
```

Bibliography

[Abr06] Y. Abramson. *Pedestrian detection for intelligent transportation systems*. EMP Press, 2006.

[Ame03] A. Amer. Voting-based simultaneous tracking of multiple video objects. In *Proc. SPIE Int. Symposium on Electronic Imaging*, pages 500–511, 2003.

[ASG05] Y. Abramson, B. Steux, and H. Ghorayeb. Yef real-time object detection. In *ALART'05:International workshop on Automatic Learning and Real-Time*, pages 5–13, 2005.

[BA04] J. Bobruk and D. Austin. Laser motion detection and hypothesis tracking from a mobile platform. In *Australasian Conference on Robotics and Automation (ACRA)*, 2004.

[BER03] J. Black, T. Ellis, and P. Rosin. A novel method for video tracking performance evaluation. *In International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance*, pages 125–132, 2003.

[BFH04a] I. Buck, K. Fatahalian, and Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. *Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–20, Aug 2004.

[BFH+04b] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for gpus: Stream computing on graphics hardware, 2004.

[BMGE01] T. Boult, R. Micheals, X. Gao, and M. Eckmann. Into the woods: visual surveillance of noncooperative and camouflaged targets in complex outdoor settings, 2001.

[BS03] H. Ghorayeb B. Steux, Y. Abramson. Camellia image processing library, 2003.

[CG00] C.Stauffer and W.E.L. Grimson. Adaptive background mixture models for real-time tracking. *IEEE Transactions on pattern analysis and machine intelligence*, pages 747–757, August 2000.

[CL04] B. Chen and Y. Lei. Indoor and outdoor people detection and shadow suppression by exploiting hsv color information. *cit*, 00:137–142, 2004.

[Cro84] F. C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, New York, NY, USA, 1984. ACM Press.

[CSW03] H. Cramer, U. Scheunert, and G. Wanielik. Multi sensor fusion for object detection using generalized feature models. In *International Conference on Information Fusion*, 2003.

[Ded04] Y Dedeoglu. Moving object detection, tracking and classification for smart video surveillance, 2004.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[DM00] D. Doermann and D. Mihalcik. Tools and techniques for video performance evaluation. *Proceedings of the International Conference on Pattern Recognition (ICPR00)*, pages 4167–4170, September 2000.

[DSS93] H. Drucker, R. Schapire, and P. Simard. Boosting performance in neural networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(4):705–719, 1993.

[Elf89] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22:46–57, June 1989.

[EWN04] Magnus Ekman, Fredrik Warg, and Jim Nilsson. An in-depth look at computer performance growth. Technical Report 04-9, Department of Computer Science and Engineering, Chalmers University of Technology, 2004.

[FH06] J.P. Farrugia and P. Horain. Gpucv: A framework for image processing acceleration with graphics processors. In *International Conference on Multimedia and Expo (ICME)*, Toronto, Ontario, Canada, July 9–12 2006.

[FM05] J. Fung and S. Mann. Openvidia: parallel gpu computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, New York, NY, USA, 2005. ACM Press.

[Fre90] Y. Freund. Boosting a weak learning algorithm by majority. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, pages 202–216, August 1990.

[FS95a] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *European Conference on Computational Learning Theory*, pages 23–37, 1995.

[FS95b] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory: Second European Conference, EuroCOLT '95*, pages 23–37. Springer-Verlag, 1995.

[FS99] Yoav Freund and R. E. Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, Sep 1999. Appearing in Japanese, translation by Naoki Abe.

[Gho06] H. Ghorayeb. Libadaboost: developer guide, 2006.

[HA05] D. Hall and Al. Comparison of target detection algorithms using adaptive background models. *INRIA Rhone-Alpes, France and IST Lisbon,Portugal and University of Edinburgh,UK*, pages 585–601, 2005.

[HBC+03] A. Hampapur, L. Brown, J. Connell, S. Pankanti, A. Senior, and Y. Tian. Smart surveillance: Applications, technologies and implications, 2003.

[HBC06] Ghorayeb H., Steux B., and Laurgeau C. Boosted algorithms for visual object detection on graphics processing units. In *ACCV06: Asian Conference on Computer Vision 2006*, pages 254–263, Hyderabad, India, 2006.

[Hei96] F. Heijden. *Image Based Measurement Systems: Object Recognition and Parameter Estimation*. Wiley, 1996.

[Int06] Intel. Intel processors product list, 2006.

[JM02] P.Perez C.Hue J.Vermaak and M.Gangnet. Color-based probabilistic tracking. *IEEE Transactions on multimedia*, 2002.

[JRO99] J.Staufer, R.Mech, and J. Ostermann. Detection of moving cast shadows for object segmentation. *IEEE Transactions on multimedia*, pages 65–76, March 1999.

[JWSX02] C. Jaynes, S. Webb, R. Steele, and Q. Xiong. An open development environment for evaluation of video surveillance systems, 2002.

[Ka04] Kenji.O and all. A boosted particle filter multi–target detection and tracking. *ICCV*, 2004.

[Kal60] E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME-Journal of Basic Engineering*, 82:35–45, 1960.

[KDdlF+04] M. Kais, S. Dauvillier, A. de la Fortelle, I. Masaki, and C. Laugier. Towards outdoor localization using gis, vision system and stochastic error propagation. In *International Conference on Autonomous Robots and Agents*, December 2004.

[KNAL05] A. Khammari, F. Nashashibi, Y. Abramson, and C. Laurgeau. Vehicle detection combining gradient analysis and adaboost classification. In *Intelligent Transportation Systems, 2005. Proceedings. 2005 IEEE*, pages 66– 71, September 2005.

[KV88] M. Kearns and L. G. Valiant. Learning boolean formula or finite automate is as hard as factoring. Technical Report TR-14-88, Harvard University Aiken Computation Laboratory, August 1988.

[KV89] Michael Kearns and Leslie G. Valiant. Cryptographic limitations on learning boolean formula and finite automate. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 433–444, May 1989.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[nVI07a] nVIDIA. nvidia graphic cards, 2007.

[nVI07b] nVIDIA. Cuda programming guide: Nvidia confidential, prepared and provided under nda, 21 nov 2007.

[OPS+97] M. Oren, C. Papageorgiou, P. Sinha, E. Osuna, and T. Poggio. Pedestrian detection using wavelet templates. In *Proc. Computer Vision and Pattern Recognition*, pages 193–199, June 1997.

[PEM06] T. Parag, A. Elgammal, and A. Mittal. A framework for feature selection for background subtraction. In *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1916–1923, Washington, DC, USA, 2006. IEEE Computer Society.

[RE95] P.L. Rosin and T. Ellis. Image difference threshold strategies and shadow detection. In *Proc: 6th BMVC 1995 conf.*, pages 347–356, 1995.

[RSR+02] M. Rochery, R. Schapire, M. Rahim, N. Gupta, G. Riccardi, S. Bangalore, H. Alshawi, and S. Douglas. Combining prior knowledge and boosting for call classification in spoken language dialogue. In *International Conference on Accoustics, Speech and Signal Processing*, 2002.

[SAG03a] B. Steux, Y. Abramson, and H. Ghorayeb. Initial algorithms 1, delivrable 3.2b, project ist-2001-34410, public report, 2003.

[SAG03b] B. Steux, Y. Abramson, and H. Ghorayeb. Report on mapped algorithms, delivrable 3.5, projet ist-2001-34410, internal report, 2003.

[Sch89] R. E. Schapire. The strength of weak learnability. In *30th Annual Symposium on Foundations of Computer Science*, pages 28–33, October 1989.

[SDK05] R. Strzodka, M. Doggett, and A. Kolb. Scientific computation for simulations on programmable graphics hardware. *Simulation Modelling Practice and Theory, Special Issue: Programmable Graphics Hardware*, 13(8):667–680, Nov 2005.

[SEN98] J. Steffens, E. Elagin, and H. Neven. Person spotter-fast and robust system for human detection. In *Proc. of IEEE Intl. Conf. on Automatic Face and Gesture Recognition*, pages 516–521, 1998.

[SS98] R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, pages 80–91, 1998. To appear, *Machine Learning*.

[TDD99] T.Horprasert, D.Harwood, and L.S. Davis. A statistical approach for real-time robust background subtraction and shadow detection. *Proceedings of International Conference on computer vision*, 1999.

[TKBM99] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers. Wallflower: Principles and practice of background maintenance. In *International Conference on Computer Vision (ICCV)*, volume 1, pages 255–261, 1999.

[Val84] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[VJ01a] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *European Conference on Computational Learning Theory*, 2001.

[VJ01b] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition*, pages 511–518, 2001.

[VJJR02] V.Y.Marianoand, J.Min, JH.Park, and R.Kasturi. Performance evaluation of object detection algorithms. *In International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance*, pages 965–969, 2002.

[VJS03] Paul Viola, Michael J. Jones, and Daniel Snow. Detecting pedestrians using patterns of motion and appearance. In *IEEE International Conference on Computer Vision*, pages 734–741, Nice, France, October 2003.

[WATA97] C. Wren, A.Azarbayejani, T.Darrell, and A.Pentland. Pfinder:real-time tracking of the human body. *IEEE Transactions on pattern analysis and machine intelligence*, 19:780–785, July 1997.

[WHT03] L. Wang, W. Hu, and T. Tan. Recent developments in human motion analysis. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition*, page 585601, 2003.

[WWT03] L. Wang, W.HU, and T.Tan. Recent developments in human motion analysis. *Pattern Recognition*, 36:585–601, March 2003.

[YARL06] M. Yguel, O. Aycard, D. Raulo, and C. Laugier. Grid based fusion of offboard cameras. In *IEEE International Conference on Intelligent Vehicules*, 2006.

[ZJHW06] H. Zhang, W. Jia, X. He, and Q. Wu. Learning-based license plate detection using global and local features. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, pages 1102–1105, August 2006.

Résumé

Notre objectif est d'étudier les algorithmes de vision utilisés aux différents niveaux dans une chaîne de traitement vidéo intelligente. On a prototypé une chaîne de traitement générique dédiée à l'analyse du contenu du flux vidéo. En se basant sur cette chaîne de traitement, on a développé une application de détection et de suivi de piétons. Cette application est une partie intégrante du projet PUVAME.

Cette chaîne de traitement générique est composée de plusieurs étapes: détection, classification et suivi d'objets. D'autres étapes de plus haut niveau sont envisagées comme la reconnaissance d'actions, l'identification, la description sémantique ainsi que la fusion des données de plusieurs caméras. On s'est intéressé aux deux premières étapes. On a exploré des algorithmes de segmentation du fond dans un flux vidéo avec caméra fixe. On a implémenté et comparé des algorithmes basés sur la modélisation adaptative du fond.

On a aussi exploré la détection visuelle d'objets basée sur l'apprentissage automatique en utilisant la technique du boosting. Cependant, On a développé une librairie intitulée LibAdaBoost qui servira comme un environnement de prototypage d'algorithmes d'apprentissage automatique. On a prototypé la technique du boosting au sein de cette librairie. On a distribué LibAdaBoost sous la licence LGPL. Cette librairie est unique avec les fonctionnalités qu'elle offre.

On a exploré l'utilisation des cartes graphiques pour l'accélération des algorithmes de vision. On a effectué le portage du détecteur visuel d'objets basé sur un classifieur généré par le boosting pour qu'il s'exécute sur le processeur graphique. On était les premiers à effectuer ce portage. On a trouvé que l'architecture du processeur graphique est la mieux adaptée pour ce genre d'algorithmes.

La chaîne de traitement a été implémentée et intégrée à l'environnement RTMaps.

On a évalué ces algorithmes sur des scénarios bien définis. Ces scénarios ont été définis dans le cadre de PUVAME.

Abstract

In this dissertation, we present our research work held at the Center of Robotics (CAOR) of the Ecole des Mines de Paris which tackles the problem of intelligent video analysis.

The primary objective of our research is to prototype a generic framework for intelligent video analysis. We optimized this framework and configured it to cope with specific application requirements. We consider a people tracker application extracted from the PUVAME project. This application aims to improve people security in urban zones near to bus stations.

Then, we have improved the generic framework for video analysis mainly for background subtraction and visual object detection. We have developed a library for machine learning specialized in boosting for visual object detection called LibAdaBoost.

To the best of our knowledge LibAdaBoost is the first library in its kind. We make LibAdaBoost available for the machine learning community under the LGPL license.

Finally we wanted to adapt the visual object detection algorithm based on boosting so that it could run on the graphics hardware. To the best of our knowledge we were the first to implement visual object detection with sliding technique on the graphics hardware. The results were promising and the prototype performed three to nine times better than the CPU.

The framework was successfully implemented and integrated to the RTMaps environment.

It was evaluated at the final session of the project PUVAME and demonstrated its fiability over various test scenarios elaborated specifically for the PUVAME project.